Hard-Object: Enforcing Object Interfaces Using Code-Range Data Protection

Daniel Wilkerson
David Alexander Molnar
Matthew Harren
John D. Kubiatowicz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-97
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-97.html

July 8, 2009

Hard Object: Enforcing Object Interfaces using Code-Range Data Protection

Daniel S. Wilkerson* David A. Molnar† Matthew T. Harren‡ John D. Kubiatowicz§

Abstract

We show how to utilize *code-range data protection* to enforce the private access specifiers of Object Oriented classes at runtime. We exploit the fact that code and data are often organized organized into *modules* exporting specified *interfaces*, even in non-Object-Oriented languages. We enforce at runtime the *integrity* and *simplistic privacy* of the module: its state cannot be written nor read other than through its interface. We provide module integrity even to non- memory-safe languages such as C and C++, without requiring automatic memory management. This is not best-effort protection: when used properly, we comprehensively guarantee that one software module cannot violate the integrity of another. That is, we make software objects hard.

Our extensions are simple, modest, and provide the guarantee we claim. We give simulation measurements to show the performance overhead is low. We show how most software can be compiled to take advantage of these extensions with modest and partially automatable modification.

1. INTRODUCTION

We face a crisis of correctness. Modern applications contain millions of lines of code in non-type-safe languages such as C and C++; they also reflect hundreds of man-years of effort and therefore will not be soon discarded. The sheer scope of these applications coupled with the constant presence of poorly written libraries, malicious code, and badly maintained execution environments has led to chaos—computer users expect their applications to crash, lose data, and behave unexpectedly. Further, now that they are networked and their use pervasive in our national infrastructure, the situation is critical. While techniques to verify program correctness have long been among the Holy Grails of Computer Science, such techniques seem forever out of reach (and are intractable in general).

Since general techniques are not available, programmers and system designers have developed a set of best practices to try to manage complexity in an effort to achieve correctness. Experienced programmers generally organize their projects into *modules* which are local collections of code that maintain invariants with respect to their data. By factoring their code in this way, programmers can apply local intuition to reason about the correctness of the system as a whole.

Unfortunately, one of the biggest obstacles toward correctness is the lack of protection between different elements of the system, thereby allowing violation of one module's invariants by another. The danger faced when different processes interfere with one another is well known. More subtle problems arise when wild pointer references overwrite arbitrary data and violate invariants *within a process*. This problem is illustrated by Figure 1: a C++ class `A` with no inheritance and only private data. The language policy is that only methods of `A` may access private data of an object of class `A`. Although the compiler can enforce this policy *of modules that it compiles* it cannot enforce them of other modules, such as that containing function `evil` in Figure 1. Along a similar vein, stack overflow and incorrect stack manipulation can lead to unexpected behavior, including well documented buffer-overrun security exploits. These problems are pernicious because they couple disparate parts of the system and defeat attempts to reliably apply local understanding to components.

```
class A {
  private: int vulnerable;
}

// ... meanwhile in another module
void evil(A *a) {
  int *x = (int *)a;  // violate the type system
  *x = corrupt_value; // corrupt 'a->vulnerable'
}
```

Figure 1: A runtime violation of module integrity.

Language designers, hardware architects, and writers of operating systems have all chipped away at this problem from different angles. Language designers have produced type-safe, verifiable languages such as Java that place restrictions on the flow of information within a program; unfortunately, these languages all seem to face a variety of performance issues and are not widely utilized. Further, the volume of legacy code is quite high. At the hardware level, computer architects have introduced the distinction between kernel and user in combination with page-level protection. With support from the operating system, such mechanisms can enforce *process-level protection*: processes are protected from one another at the page level [25, 5, 17, 29]. With the exception of Mondriaan Memory Protection [35], the overhead associated with changing address spaces or permissions makes use of such protection feasible only at course granularity. More importantly, existing page-level protection mechanisms assign process-wide permissions to each page and thus fail to defend threads from one another and further fail to defend modules within a thread from other modules.

1.1 Code-Range Data Protection

In this paper, we show how to exploit a simple hardware mechanism called *code-range data protection* to harden the interfaces between modules. Code-range data protection associates (using the page table) each page of data with a range of instructions ("code range") that are said to "own" the data. We say the "current stack frame" is data that lies

*daniel.wilkerson@gmail.com
†dmolnar@gmail.com
‡mattharren@gmail.com
§kubitron@cs.berkeley.edu

within the range delimited by the value two special stack pointer registers that point respectively to the top of the current stack frame and the bottom of the current stack. A data access by an instruction is allowed either

- when the program counter is in the owner range of the data *or*
- when the data is in the current stack frame.

Returning to our example, the variable `vulnerable` would be annotated with a range of addresses that represent the methods of class `A`. When the function `evil` attempts to change the variable `vulnerable`, the hardware will detect that the program counter is not in the correct range and disallow the access.

As we will show, when combined with support from the compiler and loader, code-range data protection can provide both *module integrity* (module state can be modified only through its interface) and *simplistic module privacy* (module state can not be read directly by other modules). Such guarantees can be provided for languages such as C and C++ without requiring automatic memory management. Of course, programmers can still design incorrect programs; we merely show how to harden object interfaces. Since this combination of techniques hardens the interface of an object oriented design methodology, we call this the "Hard Object" system.

To emphasize the main point: we are *not* advocating some hardware modifications that will make some bugs or attacks merely less likely or more inconvenient, what some might call a "best effort" improvement to computer reliability or security. In a Hard Object system, a module engineer can design the data, functions, and invariants for a module and *no other module can violate those invariants.*

1.2 Hardening Object Interfaces

Hard Object factors the process of verifying software into three parts as follows.

1. *Local module correctness:* The module author must ensure local correctness, possibly with the help of the compiler, unit testing, or a theorem prover. If the module is small, this piece of the problem is tractable.
2. *Module integrity and simplistic privacy:* Given local correctness, these properties allow us to then verify only the interfaces between modules. Without integrity and privacy guarantees, we would have to verify the entire body of code as a whole.
3. *Higher-level design patterns:* Software architects can build further correctness abstractions upon the foundations of module integrity and simplistic privacy. We show two examples, *sandboxing* and *access control lists*, in Section 4.

We concentrate our effort on ensuring the second part of these three: module integrity and simplistic privacy. Ensuring module integrity has long been recognized as a method for improving the security and reliability of computer software [5, 31]. We further conjecture that partitioning into small modules may be what makes proving programs correct finally possible.

Modularity goes far beyond programs written explicitly in object-oriented style. Many pieces of software exhibit a modular design, in which different pieces of code form *modules* separated by a well-defined interface; for example, the apache web server defines a module interface between the main program and code implementing additional features. Each module may contain data which is hidden behind the interface and should not be exposed to code outside the module. Furthermore, Zhou et al. provide empirical evidence via memory tracing that in many applications, even those not explicitly written in an modular style, a given piece of data is typically visited by only a small piece of the program's code during "normal" execution [36].

In all these cases, the unifying theme is that some pieces of data are "private": they should only be accessed by a specific, relatively small piece of the program's code. Private data should only be modified by code within the module that "owns" the data. This is exactly the guarantee provided by module integrity. Because the entire program's code typically runs in the same address space, however, a bug in any other piece of the program can potentially change the value of a given module's private data and so violate module integrity. This is the key problem we address.

1.3 Our Contribution

In the following sections, we show how to harden the interfaces for objects utilizing code-range data protection. With simple hardware extensions, a special compiler, and a load-time verifier, we can ensure that a module will not have its interfaces violated. This guarantee proceeds from three simple ideas:

1. Heap-allocated objects are marked as owned by specific modules. As a result, only authorized methods are allowed to modify them. The integrity bit is used to track the reliability of data invariants during the transfer of ownership.
2. Stack-allocated data is restricted to the currently executing frame via another type of code-range protection. In addition, a simple load-time verifier checks that any loaded program follows a stack discipline which ensures the safety of function calls.
3. Control transfer is carefully constrained so that functions are only entered at appropriate entry points. A simple load-time verifier ensures that all programs install checks to enforce this at runtime.

In later sections, we formalize these and other properties, argue why they harden the object interfaces, and show how to achieve them in greater detail. Hard Object consists of the complete system of these hardware, compiler, and verifier techniques.

This technique can be compared in spirit to the Software Fault Isolation (SFI) technique of Wahbe et al. [31]. SFI uses binary rewriting to augment control flow and data flow in a program with runtime checks that guarantee module integrity. For example, SFI might insert a runtime check ensuring that a load address is within a specific range. A key drawback of SFI is the overhead incurred by software runtime checks. Wahbe et al. report a 3× slowdown on a range of applications. While more recent systems utilizing dynamic binary translation reduce this overhead [19], some programs still exhibit substantial overhead.

User-Accessible fields of Page Table Entry

Bit `text`: Page is executable.
Bit `pageIntegrity`: Page integrity verified.
Range `pageOwner`: Code-range ownership.

User-Level instructions to manipulate PTEs

`setPageOwner(`*Pointer* `page, `*Range* `newOwner)`
`getPageOwner(`*Pointer* `page)`
`setIntegrity(`*Pointer* `page, `*Range* `newIntegrity)`
`getIntegrity(`*Pointer* `page)`
`branchIntegrityS(`*Pointer* `page, `*Address* `target)`
`branchIntegrityC(`*Pointer* `page, `*Address* `target)`

Stack Protection Mechanism

Register `framePointer`: Top of the frame.
Register `bottomOfStack`: Maximum extent of the stack.

New user-level instructions to load, store, and move `framePointer` and `bottomOfStack` between memory and normal registers.

Figure 2: Architectural elements consists of additions to page table entries (for code-range data protection), instructions to manipulate this state, and new registers (to protect elements of the stack). Note that `setPageOwner` and `setPageIntegrity` are only available to the `pageOwner`.

Our insight is that by extending the hardware memory protection mechanism slightly, we can push many of the checks performed by SFI into hardware. Consequently, our hardware extensions form a base on which we can build software mechanisms that ensure module integrity. We expand on this in Sections 2.3 and 3, in which we describe a trusted load-time verifier for binary code as part of an assurance argument for Hard Object. The verifier checks certain safety properties statically before allowing code to load. We argue that these properties, together with dynamic hardware code-range data protection, provide module integrity. That is, if the verifier passes a binary program, then the program will enjoy the module integrity property at runtime. Assuming the operating system and loader work correctly, the result is guaranteed module integrity, or "Hard" Objects.

The remainder of this paper is as follows: Section 2 describes the hardware and software mechanisms required for Hard Object. Then, Section 3 formalizes the six properties needed by the Hard Object System. Next, Section 4 describes how Hard Object can be utilized in a variety of circumstances. Then, Section 5 evaluates the performance impact of the compiler transformations required to achieve the properties of Section 3. Section 6 discusses related work.

2. MECHANISM

Hard Object consists of a combination of mechanisms in the hardware, operating system, and compiler. In this section, we will merely introduce the mechanisms and give a general idea of how they are utilized. Then, in Section 3, we will formalize the set of properties needed to enforce object interfaces and show how to achieve these properties with our mechanisms.

H-ACCESS: On a user data access, these checks are made.
- If the instruction owns the target, ALLOW.
- If the target is in frame, ALLOW.
- Otherwise, FAULT.

H-OWNER:
- Only the page owner can set page's integrity or owner.
- On an ownership change the integrity bit is cleared.

H-EXECUTE:
- Only text pages are executable.

Figure 3: Hardware rules. We say that a page's data is *owned* by the page's *pageOwner* text and that data between the *framePointer* and the *bottomOfStack* is *in frame*.

2.1 Basic Code-Range Protection Mechanism

The basic architectural elements of Code-Range Data Protection are illustrated in Figure 2. Code-Range Data Protection involves annotating each page table entry (and thus each page) with a `pageOwner` that designates the range of instructions that "own" the page and can manipulate it. A range of instructions can be thought of as a pair of pointers delimiting a contiguous region of memory. Note that an actual implementation merely needs to designate valid sets of instructions within the executing program and thus can consume significantly less space than two complete pointers.

In addition to being checked against page table protections in the usual way, every load and store is checked against the `pageOwner` field of the PTE. If the instruction address of the load or store is within the `pageOwner` range, it is allowed; otherwise it is FAULTED. Basic Access semantics are detailed in Figure 3.

Two additional bits are associated with every page as well. The `text` bit which distinguishes executable pages from data pages and the `pageIntegrity` bit which designates pages whose data integrity have been checked by the current owner.

As also shown in Figure 2, we have added new instructions to manipulate the new PTE fields. Only the `pageOwner` may set these fields. In addition, the `pageIntegrity` bit is automatically cleared whenever the `pageOwner` is changed. Since the `Integrity` bit will be checked often we also create instructions to branch on the state of the this bit.

To provide stack protection, we introduce two user-level registers with the following semantics:

1. a *framePointer* register that points to the top of the current stack frame; it is saved on the stack during a function call and saved in the thread control block during a context switch.

2. a *bottomOfStack* register that points to the maximum allowed extent of the stack; it does not change over the lifetime of a thread (Note that this is not the "stack pointer" which points to the bottom of the frame.)

We also require some instructions to load and store these registers; at minimum we require instructions to move data between the rest of the register file and the new registers.

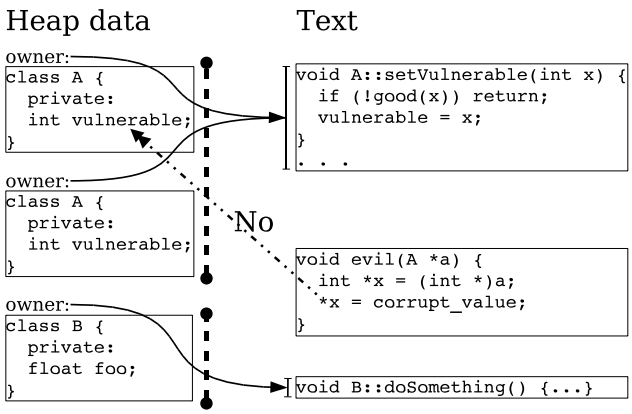

Figure 4: The object protection feature preventing the function `evil` from violating the integrity of an object of class `A`. On the left we see some heap pages and on the right some text pages. The *owner* field on each heap page points to the text range that owns it; there can be more than one object per page (owned by the same owner) but this is not shown. The double-headed dashed line indicates a failed access by a function to a heap page that it does not own. The access fails because it comes from a text address that is not in the owner range of the heap page containing the data that the access targets.

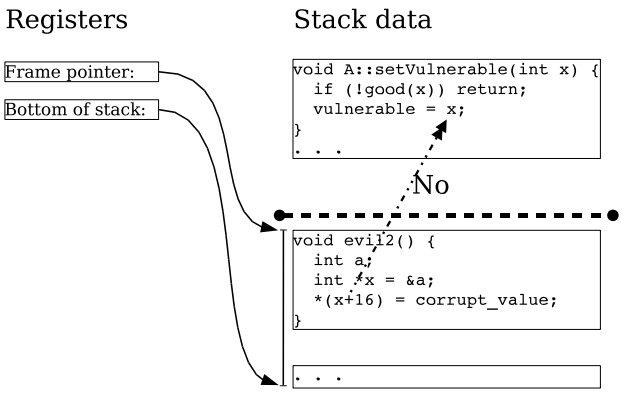

Figure 5: The function protection feature preventing the function `evil2` from violating the integrity of the stack frame of function `A::setVulnerable`. On the left we see some CPU registers and on the right some stack data pages. The *frame pointer* and *bottom of stack* registers delimit the currently accessible range of the stack. The double-headed dashed line indicates a failed access by the function `evil2` to the stack frame of `A::setVulnerable`. The access fails because it targets an address not in the currently accessible range.

2.2 Utilizing the Hardware Mechanism

At the heart of the Hard Object technique is the combined protection of *object* data and *methods*. Section 3 and Appendix A carefully details the properties that the system must possess in order to truly harden interfaces. To achieve these properties, we utilize a *verifier* at load time to make sure that the objects with which we are interacting adhere to proper stack discipline and control-flow constraints. Any *compiler* that produces code that adheres to these rules can be utilized. Although we leave the details to Section 3, we wish to illustrate how the basic hardware mechanisms described are utilized.

Object Protection: To harden the interfaces of an object, we wish to prevent the corruption of the heap state of an object. When we allocate new objects, we do so by associating blocks of memory with the code that is allowed to operate on it. This association happens, for instance, when `malloc()` returns data to use. Once the `pageOwner` field is set, any accesses to this data by non-owning code is prevented as shown in Figure 4. Note that we make use of the fact that it is possible to locate contiguously all functions ("methods") of a class.

After initializing objects appropriately, we can set the `Integrity` bit to indicate that invariants have been established. The `Integrity` bit can help us to distinguish, among other things, empty blocks of memory (returned from `malloc`) from blocks of data that we have already initialized. Further, as shown in Section 3, this bit helps us to track objects that are potentially corrupted after they change ownership.

Function Protection: Any solution that hardens object interfaces must be able to protect the methods of this object. First and foremost, methods have well defined entry points that must be respected (if functions call into the middle of methods, nothing can be guaranteed). As we discuss in Section 3, we provide a calling convention that allows the load-time verifier to guarantee that any methods are only called via their intended entry points.

Once executing, the state of a method consists of data within registers and on the stack. In order to protect methods from one another, we must protect this stack state from being corrupted by other methods. We utilize a combination of hardware and software to do this, as shown in Figure 5. At any one time, the stack frame for the active method is available between the `framePointer` and the `bottomOfStack`. All frames for calling methods are protected from modification because they lie above the `framePointer`. Further, the `bottomOfStack` register provides protection in a multithreaded environment.

Once again, we rely on the load-time verifier to guarantee that the `framePointer` register is manipulated correctly during the calling sequence to protect data from all but the active frame.

2.3 The Compiler and Verifier

The system described above places certain constraints on the compiler: The compiler must generate code that follows the architectural and verifier rules. For instance, code must be generated with the proper calling sequence to maintain the rigid boundaries of the Hard Object. Further, stack allocated objects are disallowed and must be placed on the heap instead (though for C++ objects the destruction semantics can of course remain the same). See Section 5 for more detail of the program analysis done in our evaluation implementation; space constraints prevent us from repeating it in more generality here.

Our system requires that a load-time verifier check that the compiler has obeyed certain conventions, most notably those involving control-flow. We have not yet implemented this verifier, but the analysis will be very similar to previous

Function protection

- P-ATOMIC: A function is called only at the top and does not return until it chooses to.
- P-FRAME: The stack frame of a function cannot be accessed by another function.
- P-KITH: A callee knows who its caller is.

Object protection

- P-KIN: An M-function can verify that a pointer refers to a properly-constructed M-object.
- P-OBJECT: An M-object cannot be accessed by a function not in M.
- P-FREE: Only M can destroy an M-object.

Figure 6: The computational properties provided by Hard Object. An *access* means a user-mode read or a write. An *M-object* is an object for which module M provides a constructor function.

work on enforcing "control-flow integrity" in binary code [4]. However, enforcing control-flow integrity in Hard Object is easier than in previous work because our hardware protects the integrity of return pointers on the stack.

In addition to control flow, the verifier checks properties such as the requirement that any `setPageOwner` instruction be given valid arguments (see V-PARTITION and V-PARTITION-DYNAMIC in Section 3). Because these properties are "local" and do not require whole-program analysis, they will be easy to verify.

3. ENFORCING MODULE BOUNDARIES

The goal of Hard Object is not some kind of "best effort" protection but a comprehensive *guarantee* of module *integrity* and (simplistic) *privacy* throughout the lifetime of its computation.

The proper form of such a guarantee is a proof that the boundaries of a module cannot be violated at runtime. The core of this proof is the set of six Hard Object properties given in Figure 6. In Appendix A we consider in detail the checks that must be done by the hardware, verifier, system software, and the module's own compiler (but not the compilers of other modules) in order to ensure these six properties. In this section we given an argument that once these six properties have been ensured that boundaries of a module cannot be violated.

To approach this problem we again we consider modules from an Object Oriented perspective. An object c_0 of a class C has four basic stages of its life cycle: *Construction*, *Duration*, *Operation*, and *Destruction*.

In support of guaranteeing protection throughout the object life-cycle, Hard Object ensures the specific *properties* of runtime computation given in Figure 6; properties are distinguished by a P- prefix. We review in detail the object life-cycle and discuss how these properties are sufficient to protect it throughout.

- CONSTRUCTION: Only the code for class C can construct an object c_0 from raw memory; at construction time the state of c_0 can be set to respect whatever invariants C wants to rely upon. When a module obtains a page from malloc, the module should be assured that the page is its own and no other module can modify that page. This is a simpler version of a property we call P-KIN, defined in Figure 6. The properties P-ATOMIC and P-FRAME ensure that the constructor completes as expected.
- DURATION: Once c_0 is created, no code outside of C can directly read or write its internal state, the memory allocated to it during construction. Property P-OBJECT ensures that while the functions of the class are not operating on the object, no one else is either.
- OPERATION: Code outside of C may call functions in C; however these calls are atomic: once a function is called its stack frame cannot be interfered with and it cannot be terminated abnormally. When c_0 is passed to a method of class C, there must be some way to distinguish unequivocally objects of class C from objects of other classes. Distinguishing c_0 from other objects of class C is also necessary. In Object Oriented languages there is a feature that a function that can only be called from code in the same class is called *private*; that is, a function must be able to screen its calls. Property P-KIN ensures that when a function of class C gets one of its own objects back as an argument, it recognizes it. Again, P-ATOMIC and P-FRAME ensure that the function completes as expected. P-KITH ensures that a function can refuse to operate when called from disallowed callers.
- DESTRUCTION: Only the code for class C can destruct an object c_0, returning it to raw memory; when it does it can finalize any invariants on the contents of c_0. Property P-FREE guarantees that until the class destructor destroys an object, no one else can. The usual protections cited for operations protect the process of destruction.

4. DESIGN PATTERNS

The software design patterns admitted by the system are a good measure of its usability. We consider some existing design patterns in order to demonstrate that they still work in Hard Object. We then present some new design patterns that show the power of Hard Object.

4.1 Existing patterns

Shared objects: Modules share objects by transferring ownership of the object, then validating that the transferred object satisfies module invariants. For example, module A can call a read() method in a different module B as follows. First, A allocates an array on the heap, then transfers ownership to read(), and finally calls read(). Then read() fills the array and transfers ownership back to A. The integrity bit of the array is cleared until A checks for itself that the array satisfies the invariants A expects from read().

We note that A must mark the array with metadata indicating that it is the result of a call to read(); otherwise, read() can instead return a page of corrupted objects to A and module B can mount an attack on module A as follows. Upon the return of the page to the ownership of A the integrity bit will be cleared, but A must set it again as A needs to use the data; later module B calls a method on A, passing to A a pointer into the page of corrupted objects if it where one of A's objects. The lesson is that if a module is

managing several kinds of shared objects, each kind must be distinguished via metadata, either in the objects themselves or on a separate page managed by A.

Inheritance: We say class A *boxed-relates* to class B if a member of an instance a_0 of A *points to* an instance b_0 of B. In contrast, A *unboxed relates* to B if the data of b_0 is embedded into a_0. The unboxed idiom is the typical, but not required, implementation of both C++ inheritance and embedded member objects. When using Hard Object an unboxed relationship requires that A and B reside in the same module. Therefore, the simplest way to implement a class hierarchy is encapsulated within a single module. If we want to partition a hierarchy across modules, we switch some relationships to to boxed-relations.

If a subclass boxed-relates to a superclass, however, the client cannot call any public superclass methods directly. Instead, the subclass must provide stub-methods which both 1) indirect through the pointer to the superclass data and 2) delegate to the corresponding superclass method.

A subclass that unboxed-relates (embeds into) to its superclass complicates validation of 'this' pointers in the superclass methods because the superclass objects may have a different size than the subclass objects and yet the superclass methods are expected to be able to operate on either; see Section A.5 for techniques for handling this situation.

Stack-allocated objects form an unboxed relationship with the stack frame "object." We must always transform this to a boxed relationship by moving the object to the heap. The resulting objects can be destructed at block exit, just as in the standard C++ semantics.

Virtual dispatch: In a Hard Object system, a client of a polymorphic object A in another module C cannot access A's vtable directly to invoke an `A::foo()` method. Instead, A can manage its own vtable accesses. The call site `a_0->foo()` in C is now compiled to call a new `A::__dispatch()` function. The function name "foo" is compiled to an offset into the vtable, then passed as an argument. Then `A::__dispatch()` can access the vtable of the class hierarchy and jump to the appropriate method.

Inline functions: Functions may be inlined within a module, but not otherwise.

4.2 New patterns

Nested Modules: Hard Object modules may nest hierarchically. For example, one may design a module M composed of sub-modules M1 and M2. Each sub-module is constructed in the standard way, where the data of module M1 is protected from the code of module M2 and vice versa. However, module M may also have its own "M-global" data not part of one of its sub-modules, as well as code to manipulate that data, and wish to also allow access to that data from modules M1 and M2. This arrangement is easily accomplished by laying out the code for modules M and sub-modules M1 and M2 contiguously in memory. The code-range for M1's data is set to the code of M1, and similarly for M2. However, the code-range for M-global data is set to include the code for M *and* its sub-modules M1 and M2. Using this hierarchical design pattern an engineer need not choose a *particular* granularity at which to partition the program, but may use Hard Object protections at multiple levels of granularity.

Lightweight Recovery-Oriented Computing: A common pattern for constructing complex Graphical User Interfaces (GUIs) is to separate them into a UI and a Model [26]. The UI is often complex and therefore prone to bugs, whereas the Model is usually simpler yet holds the essential data. This software separation is not enforced in hardware in most programs: a bug in the UI can corrupt the Model and cause the whole program to crash. Many web applications therefore separate these layers using different processes for, say, the web server and the database; however this solution is heavyweight.

Using Hard Object the UI and Model are easily separated while residing in the same process, making this technique much more widely applicable. Imagine the following scenario in the spirit of Recovery-Oriented Computing[24]. The UI erroneously attempts to write to a Model page. Under Hard Object the process faults to a handler; the handler knows the Model is not corrupted since hardware protects it, so it just reboots the UI. What would have been a program crash becomes simply a screen flash as the UI redraws.

Access Control: The integrity of module A may depend on the state of other objects outside of its module. The hard ownership property provided by Hard Object will not protect those other objects, however we may build a software notion of ownership on top of the Hard Object guarantees as follows. Suppose module A has members pointing to objects of another module, L, a trusted library module separate from A. A different, untrusted, module U cannot corrupt the internal state of an L-object but could still operate on the L-object by calling methods. We would like to prevent such access; more generally, we would like to provide an *access control list* for L that specifies access policies for interfaces of L.

Access control for module interfaces can be implemented on top of Hard Object by storing metadata with objects, then extending methods to check this metadata. We sketch a simple example. Suppose L wishes to restrict access only to a class A. Then, the L-constructor annotates each new L-object l_i with a *softOwner*, the Range pageOwner of the module A. Methods of L at runtime check the softOwner of l_i. The methods then refuse to operate on l_i unless the call comes from the softOwner, in this case A. By extending this soft-ownership metadata, we can support arbitrary access control policies.

Sandboxing: We can implement sandboxing by specifying a reserved *sandbox* area of memory. Module A downloads untrusted module U into the sandbox and allows itself to be called from U. Other modules reject calls from the sandbox. If module A loads untrusted module U, say from the network, A may tell the system that any faults in U should return to a U-designated handler in A. We augment the verifier to ensure that U contains no system calls. Therefore, the module U must call through module A to perform any system calls; A can enforce an arbitrary access policy for U.

Project	Heapify	Other Auto	Manual	Total LOC
apache	270	279	166	83,301
cron	23	84	179	2,349

Figure 7: Code changes in apache and cron. The first two columns measure automatic changes, while the third reports the number of lines of code changed by hand.

5. EVALUATION

We believe the main cost of Hard Object will be in the changes software must make to take advantage of Hard Object protections. To measure this cost we chose two real programs written in C and compared their performance (1) when using our emulation the Hard Object compilation and execution process versus (2) when using the standard compilation and execution process. For our benchmark programs we chose apache 1.3.34 and vixie-cron 3.0.1, both of which have had security-critical bugs [30, 1]. For each case study we partitioned the program program into Hard Object modules, performed static-time dataflow analyses of the program, applied a restricted version of the Hard Object compiler transformation to the program, and then benchmarked the results on standard IA-32 hardware.

Choosing a Partition into Modules. We partitioned each program into Hard Object modules at the file granularity. Recall that Hard Object modules can include multiple program classes or modules.

- For apache, we leveraged the fact that the code is pre-partitioned into modules; we chose to separate the source file `mod_cgi.c` from the rest of the apache code.
- For cron, we observed that a single function, `load_entry`, performs parsing of crontab entries; because a crontab file is under the control of a possibly malicious user, we chose to separate this function and its helper functions from the rest of the code.

Static Analysis. We performed two different static-time dataflow analyses of the programs to be evaluated.

1. We verified that no data allocated by one module can ever be accessed by another module. If these accesses had been allowed, on a real Hard Object system a hardware check would have failed.
2. We inferred which functions are *public*: they can be called from another module. Such functions must have runtime checks C-INTEGRITY and C-TYPE checks described in Section A.5 added at the top in order to fully protect themselves from other modules.

To verify the property of (1) we augmented Cqual++, a polymorphic dataflow analysis tool [3], to infer and check module boundaries. Note that we say that code and data has the "color" of the module of the file in which it was defined. An *access site* is a point in the program where code accesses data. An access site is valid at runtime only if the code and data are the same module color. The dataflow analysis then tells us whether an access site with two incompatible colors could occur at runtime. If so, this indicates an illegal module crossing, and we issue a warning. In the case of a warning the code must then be changed by, say, adding an accessor method for the data to its module and then having the other module call that accessor at the call site rather than accessing the data directly. If we got a warning and had not modified the code to remove it then if the program had been run on real Hard Object hardware a hardware access check would have failed at runtime (if the code fragment at that particular warning site had been executed).

To infer the locations of (2) where the C-INTEGRITY and C-TYPE runtime checks must be added, we first determine whether a function could ever be called from across a module boundary; if so, we call it *public*, otherwise we call it *private*. We can infer module boundaries easily in the case of direct function calls, and we conservatively assume that a function which has its address taken could be called through a function pointer from across a module boundary.

Note that as an optimization we can be sure that a function which has its address taken is never called from across a module boundary if no code outside the module ever jumps through a function pointer. Unfortunately at the time of this writing we can no longer be sure if we uses this optimization or not[1]. While this optimization could possibly be used in small programs with only few modules, in large programs with many modules that optimization is not likely to be applicable and so we should not have (possibly) used the optimization as it could have made our numbers look better than one would obtain in general.

Note that the correctness of our static analysis depends on the assumptions that `const` is honored and that casts to and from `void*` are not used to do a hidden cast. We did not verify that these assumptions hold everywhere. Please note that in our experimental system we trusted the compiler's analysis of the whole program; however in a real Hard Object system a module author need not trust trust the compiler of other modules, only of their own module (as well as other operating system components).

A Mini Hard-Object Compiler. As much as feasible we made the code do operations and checks it would have to do if it were part of a real Hard Object system.

We reduce the manual intervention needed for Hard Object by using the CIL [22] tool to automatically perform certain source-to-source transformations, as follows. First, all stack data that has its address taken is allocated on the heap instead, to allow references to these objects to be passed to other functions. Second, we replace calls to malloc and related functions with a malloc that performs dummy operations to simulate the cost of verification. Third, we insert getters for cross-module data access. Finally, we insert calls to a module-local verifier function at module boundaries to enforce the P-KIN property.

We faked the overhead of manipulating some of the nonexistent hardware elements, such as the integrity bit, by instead reading or writing a volatile memory location.

[1]This article was originally written immediately after the experiments were performed. The final editing pass before publication was done a few years later and this ambiguity was only noticed then. No one can now remember the answer. We have no desire to attempt to get the old code working again and re-run those experiments. Therefore we are reporting what we have.

Experiment	Mean	Std	n	Ratio
flat-1-normal	2.1	1	5000	
flat-1-xform	2.1	0.8	5000	1.0
flat-2-normal	2.1	100.4	5000	
flat-2-xform	2.1	115.7	5000	1.0
flat-8-normal	2.0	31.5	5000	
flat-8-xform	2.1	90.8	5000	1.1
cgi-1-normal	42.3	2.6	5000	
cgi-1-xform	42.4	2.1	5000	1.0
cgi-2-normal	42.2	179.9	5000	
cgi-2-xform	42.3	173.0	5000	1.0
cgi-8-normal	42.4	820.0	5000	
cgi-8-xform	42.4	1052.5	5000	1.0
cron-normal	0.23	0.001	13	
cron-xform	0.32	0.002	13	1.4

Figure 8: Performance results. Here "-xform" indicates a transformed program. Trailing numbers for the flat and cgi experiments indicate concurrency levels. Times are in seconds. The "ratio" column reports the overhead observed in each pair of experiments. All programs were compiled with "-O2."

These transformations are a restricted version of those that would be performed by a fully Hard-Object-aware compiler; we detail the gaps at the end of this section.

Hand Partitioning. If CIL skips a transformation that is too complicated to do automatically (such as validating an array of pointers), the Cqual analysis will detect the problem and tell us where to intervene manually. For example, in apache we added four new helper functions. A summary of our changes appears in Figure 7.

Benchmarks. We then benchmarked each program against its transformed version. For apache, we used the apache benchmark tool ab to perform five thousand requests to both a transformed and untransformed apache running on localhost. We did two separate experiments; the first served the apache default HTML page only, while the second invoked a simple "Hello World!" CGI script written in Perl. For cron, we repeatedly loaded a 232K crontab file and measured the time to parse the file. We show performance results in Figure 8: "flat" refers to the default HTML page experiment, while "cgi" refers to the CGI experiment. The "-xform" denotes a transformed program. We also experimented with different levels of concurrency provided by ab, which are indicated by numbers in the table. For example, the row "flat-8-xform" refers to the flat file test on a transformed apache with eight concurrent requests. All experiments were carried out on a 1.6 GHz Pentium M system with 1 GB of RAM under Fedora Core 3 2.6.11.7-lc1.

Results. Our results show negligible overhead from our transformations in the flat file and the cgi apache benchmark. While we had to modify a small amount of code by hand for correctness, we made no modifications specifically to improve performance. We attribute these results to the fact that apache is already separated into software modules; our work simply checks this separation. In particular, module crossings do not appear as part of any inner loop.

In contrast, we modified cron by hand for performance. We discovered stack-allocated string buffers within the entry.c module passed to other functions but never escaped outside the module. Our transform moved these to the heap, incurring several malloc calls inside a loop. We expect a more sophisticated compiler could detect and fix such buffers automatically.

Even so, cron exhibits a 40% overhead with our modifications. An examination of the code shows that the entry.c module must make frequent updates in an inner loop to a data structure outside the module. As a result, we incur module crossing overhead inside the loop. It should be possible to reduce this overhead by changing the API; for example, entry.c could store its data in a temporary structure that is validated by the outside module all at once.

Un-Implemented Hard Object Support. Our transformations omitted the following features that would be present on a real Hard Object system. First, a Hard Object malloc() and brk() take a callerRange argument to use in ownership transfer of the allocated page, must perform a verifiably-correct ownership transfer of the allocated page, and allocate pages on page boundaries. Second, every function return must move the frame pointer an extra time just to read it before resetting it. While our allocator did perform work to simulate the overhead of a verify, it did not maintain separate memory pools for each module and so did not incur memory fragmentation overhead. Finally, we did not implement a new calling convention that would let us use the stack access rule to protect the saved return address. We believe these would add, in most cases, only a small performance overhead above what we measured here.

6. RELATED WORK

Intel Segmented Addressing. The Intel IA-32 architecture supports fine-grained permissions by use of segments. Gorman describes how two different segments may refer to the same data to provide protected sharing between processes [15]. Unfortunately, most operating systems today use page-based memory and run in only one or two segments. IA-32 also has *call gates* for function calls between two segments with different privilege levels; these call gates require copying, while our mechanism does not. As part of a call gate transfer of execution, the processor copies a specified number of bytes from the lower-privileged stack to the higher-privileged stack. This is similar in spirit to our mechanism for protected function calls, which exposes only a portion of the stack to the callee, but our mechanism does not require copying memory.

Okamoto / Nozue. Okamoto et al. [23] and Nozue, et al. [13] propose a memory protection mechanism that, as we do, uses the program counter value to make memory access decisions. However Okamoto et al. make it rather plain that their goal is to allow different *threads* to share the same memory space while being protected from one another (while also perhaps also allowing the sharing of heap memory in a way similar to that allowed by *mmap*), whereas our goal is to allow different *modules* to share the same memory space while being protected from one another.

This difference in goals shows up in many ways in the two designs. Their design does seem to contain hardware features sufficient to provide at least the functionality provided

by the Hard Object owner range protection, however they do so through the use of more hardware complexity than we do. Further they do not offer the Hard Object user-mode ownership transfer feature nor the user-mode integrity bit; nor do they seem to (1) allow as efficient cross-module boundary calls as we do, nor (2) offer to protect stack frames at the module granularity as we do.

Their Access Control List (ACL) system annotates memory pages with hardware ranges of text that can read and write them, just as Hard Object does; however the Nozue design contains more hardware complexity than would be needed by software designed for Hard Object hardware. Their design [13] calls for a PTE to contain an ACL "the ACL having a plurality (three in FIG. 27) of entries" and "a pointer... to next ACL indicating the address of a next ACL for the page represented by this address table entry", which suggests an unbounded linked list of Access Control Lists which must be followed to determine if an access is allowed. In contrast in the Hard Object design simply requires *one* integrity bit, and *one* owner range; any further access control complexity is expected to be performed by the software of the module through use of the P-KITH property.

In their design it seems that setting the ACLs on a page requires a call into the kernel, as no other mechanism seems to be suggested; this seems to be in keeping with their goal of thread-granularity protection. In current microprocessor architectures and operating systems, kernel calls are expensive (however they do further suggest a change to a Single Address Space Operating System where kernel calls might be cheaper) and would not be as suitable for finer-grained module-granularity ownership transfer. In contrast the Hard Object method of transferring memory ownership uses a single user-mode hardware instruction. Further the Hard Object design provides an integrity bit for protecting against adversarial ownership transfers; their design does not.

Their design also do not seem to provide any method for protecting the stack frame of a function in one module from the code in another module, or at least not in a way that would also allow for the traditional software stack organization where function arguments and return values can be passed on the stack; they only seem to provide a way for protecting the entire stack of one thread from another thread. Further, function calls across protection boundaries seem to be required to go through a call gate mechanism, seemingly requiring a double jump. In contrast Hard Object provides a hardware mechanism for protecting the stack at the frame granularity, protecting the frame of any suspended function from attack by the current function. Further we do so while requiring only very minor changes to (1) the current organization of stack frames and (2) to the way in which functions are called, except in the case of dispatching virtual methods across protection boundaries where a double jump seems to be necessary (see section 4.1).

Mondriaan Memory Protection. Mondriaan Memory Protection is a hardware system that focuses on providing memory access protection at the machine word granularity [35, 34, 33].

In the Mondriaan design a "protection domain" is a map from addresses to access permissions for those addresses, similar to the way access permissions are annotated on pages by the Memory Management Unit in a conventional system, but at a finer granularity. Each domain has its own "permissions table" which attaches "permission value" meta-data to memory addresses; the permission value of an address determines whether the current thread may read, write, and/or execute the data at the address. For a particular thread one protection domain is active at a time, as indicated by the current value of the Protection Domain ID register.

Mondriaan uses a "permissions lookaside buffer" to reduce the performance overhead of fine-grained protection; they demonstrate that this technique is effective.

The cost of swapping permissions lookaside buffers on a protection domain crossing function call is not measured; we think this cost is likely to be significant. In contrast, since in a Hard Object system permissions annotations on data need not be changed on cross domain calls, the cost of such calls is lightweight.

The Mondriaan design has a notion of the "owner" of data which it seems can give permissions to other protection domains, revoke permissions, and transfer ownership [33, section 6.5]; in contrast Hard Object conflates the notion of being an owner and having access rights. Mondriaan requires these actions taken by an owner to be done using a kernel crossing. In contrast Hard Object requires no kernel crossing to change data ownership; instead a user-mode hardware instruction is provided for this purpose.

Very similar to Hard Object, the Mondriaan design also provides a method of stack data protection using two registers to delimit the active stack, one pointing to the base of frame and the other to the stack limit.

Note that unlike the Mondriaan design, Hard Object does require additional static analysis of software to make module separation complete. We have not considered here how much simpler and more efficient the Mondriaan system could be (in particular the mechanism for performing a protection domain crossing function call) if their system also used static analysis.

Other Hardware Approaches. Several architectures, such as PA-RISC, PowerPC, and IA-64, include support for *page groups*, in which pages are associated with lists of protection identifiers [20]. The processor has several protection ID registers; access is allowed only if a page ID matches an ID in one of these registers. A key issue with these architectures is maintenance of the protection ID registers. For example, accessing five different protection domains simultaneously on PA-RISC incurs a performance penalty, as the architecture has only four protection ID registers [16]. We address this issue by using the program counter and code-range annotations, thereby allowing automatic update of the "protection ID." As a result, our approach scales to a large number of protection domains, and crossing a protection domain requires little overhead.

The Minos system uses a hardware tag to mark data from untrusted sources as "tainted." Taint flow is tracked through the system and control-flow checked to ensure that it does not depend on tainted data. Minos has been shown to prevent several control-flow attacks, but it does not address non-control-flow attacks [10].

The Nooks project uses existing virtual memory protection mechanisms to isolate device drivers. Because these device drivers have privileges to undo such protections, their work applies only to buggy, not malicious code [28]. In contrast, Hard Object protects module data even from malicious code.

Non-Control-Data Attacks. Classic attacks in computer security, such as stack overflow attacks, focus on corrupting control flow. Chen et al. [9] show *non-control-data* attacks are also a serious problem, e.g. an adversary may overwrite a configuration file for sshd to gain root privileges. The module integrity property provided by Hard Object protects non-control-data as well as control flow.

iWatcher and AccMon. iWatcher provides hardware support for registering a range of memory addresses for "watching," together with the type of access to be watched (load or store) and the address of a monitor function [37, 38]. When a watched access occurs, the processor directly transfers execution to the monitor function, which runs in the same address space and with the same privileges as the monitored program. Zhou et al. apply the iWatcher infrastructure to build AccMon, a system for automatically discovering and enforcing "PC-based invariants." [36] A PC-based invariant is a set of program counters associated with a memory; they observe independently of our work that only a small set of program counters ever visit a piece of data. AccMon dynamically derives invariants from program runs and checks for violations of these invariants.

The major difference between AccMon and our work is that we focus on providing a single property with high assurance, namely module integrity. We develop a trusted verifier and a correctness argument that give us guarantees about module integrity. In contrast, iWatcher checks a wide range of properties in the presence of buggy or malicious code. AccMon uses this infrastructure to provide "best effort" protection against memory errors, augmented with a Bloom filter for caching recent protection decisions. Furthermore, we show how compiler support can derive permission settings that are guaranteed to work for all program runs, while AccMon uses a statistical learning theory approach that may cause false positives (though these were rare in the programs tested).

Microreboots. Candea et al. propose "microrebooting": partitioning software into independently restartable modules [7, 8]. Tests on a prototype J2EE environment show that microrebooting can effectively mask failures in a web service application [8]. Hard Object provides a lightweight way to isolate modules without incurring the overhead of Java or other type-safe languages.

Binary Rewriting. There are several techniques for protecting software by rewriting binary code. The seminal work in this area is Software Fault Isolation [31] (SFI), in which binaries are instrumented with dynamic checks at load time. Abadi et al. formalize the "control-flow integrity" property and use binary rewriting to ensure that legacy applications have this property [4]. Kiriansky et al. use binary rewriting as part of "program shepherding," in which fine-grained policies regarding program control flow are embedded in a program binary [19]. The VINO operating system isolates untrusted kernel extensions [27]. These systems focus on control-flow integrity, except for SFI. All these systems incur a performance penalty for checks in software.

Source-Level Protection. StackGuard inserts runtime checks for stack-smashing attacks, while PointerGuard implements a fast pointer obfuscation method aimed at defeating control-flow attacks. While both work against a significant class of malware and require minimal changes to source code, they are only "best-effort" protections. The CCured project aims at retrofitting legacy C code to be memory-safe, thereby providing stronger guarantees [21]. Unfortunately, CCured may require significant revisions of the source code, and has a performance penalty of up to 87%.

Privilege Separation separates programs into different communicating processes; the technique can be partially automated [25, 6]. Efstathopoulos et al. have even gone so far as to propose a new operating system, Asbestos, motivated in part by the desire to make privilege separation easier for the programmer [11, 12]. With Hard Object, privilege separation within a process can be provided by changing the ownership of the appropriate heap pages.

Virtual Machines. Language-based virtual machines, such as the Java Virtual Machine or the Microsoft Common Language Runtime, offer the benefits of memory safety and strongly typed languages, but at a performance cost. Hertz and Berger estimate that the Java garbage collection means five times as much memory is required to obtain performance equivalent to a program without garbage collection [18]. In addition, legacy C/C++ software must be rewritten to take advantage of these benefits.

A different approach uses trusted virtual machine monitors, such as Terra, to provide isolation between different kernels running on the same physical hardware [14]. While this provides important benefits for applications that can tolerate coarse-grained partitioning, such as web servers, it does not allow us to partition within a single process. In contrast, Hard Object provides for isolation between different modules in the same process.

Program Verification. Finally, there is a long history of work on program verification. The vast majority of current software is not verified due to the amazing difficulty of the task. We suggest that by isolating modules from one another their correctness proofs may therefore be similarly isolated and program verification may finally become a tractable problem.

Subsequent work on Hard Object. Subsequent to the original writing of this paper, a U.S. patent application [32] has been filed, which as of fall 2008 should be obtainable from the website of the U.S. Patent and Trademark office [2]. In this paper we presented a rather simple configuration that protects heap memory only at the page-granularity. The patent application however presents more designs based on re-using some of the work of Mondriaan Memory Protection [35, 34, 33], discussed elsewhere in this section. At the cost of a bit more hardware complexity, these designs allow for much finer-grained control over the access to objects, reducing the need to alter software in order to make use of the Hard Object mechanism. Further, the patent application presents a design for eliminating the traditional heavyweight user-kernel boundary by using Hard Object protections instead, and further using Hard Object to protect parts of the kernel from other parts. We do not repeat these designs here and refer the reader to the patent application.

7. ACKNOWLEDGEMENTS

The authors gratefully acknowledge Simon Goldsmith for pointing out the flaw in an earlier design that was corrected using the integrity bit (Matt Harren independently also found this flaw), and Ann Lehman (now Ann Harren) for her help in obtaining a copy of Okamoto et. al. [23] on short notice.

8. REFERENCES

[1] Common vulnerabilities and exposures. http://www.cve.mitre.org/.

[2] United states patent and trademark office website. `http://uspto.gov/`.

[3] Website of Elsa, Oink, and Cqual++. `http://www.cubewano.org/oink`.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security*, November 2005.

[5] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. MACH: a new kernel foundation for UNIX development. In *Proc. Summer USENIX*, 1986.

[6] David Brumley and Dawn Xiaodong Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72. USENIX, 2004.

[7] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[9] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security 2005*, pages 177–192, 2005. `http://www.usenix.org/events/sec05/tech/chen.html`.

[10] J. R. Crandall and F.T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *IEEE MICRO*, 2004.

[11] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *20th Symposium on Operating Systems Principles*, 2005.

[12] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Making least privilege a right, not a privilege. In *HotOS*, 2005.

[13] Nozue et al. United states patent 5,890,189, Mar 30, 1999.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[15] S. Gorman. Overview of the protected mode operation of the intel architecture, 2005. `http://www.intel.com/design/intarch/papers/exc_ia.pdf`.

[16] E. Hamilton. Response to comp.sys.hp.hpux message 'SEVERE performance problems with shared libraries at HP-UX 9.05', November 1995. `http://groups.google.com/group/comp.sys.hp.hpux/msg/184b16c53b99511a?hl=en&`.

[17] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th Symposium on Operating Systems Principles (SOSP)*, 1997.

[18] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. *SIGPLAN Not.*, 40(10):313–326, 2005.

[19] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding, 2002.

[20] E.J. Koldinger, J.S. Chase, and S.J. Eggers. Architectural support for single address space operating systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS) V*, pages 175–186, 1992.

[21] G.C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2005.

[22] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Conference on Compiler Construction*, 2002.

[23] T. Okamoto, H. Segawa, S. H. Shin, H. Nozue, Ken ichi Maeda, and M. Saito. A micro-kernel architecture for next generation processor. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 83–94, Berkeley, CA, USA, 1992. USENIX Association.

[24] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, and Noah Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. In *UC Berkeley Computer Science Technical Report UCB/CSD-02-1175*, Berkeley, CA, March 2002. U.C. Berkeley.

[25] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, 2003.

[26] Trygve Reenskaug. Thing-model-view-editor: an example from a planningsystem, May 12, 1979. Internal document at Xerox PARC.

[27] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.

[28] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems, 2003.

[29] M.M. Swift, B.N. Bershad, and H.M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), November 2004.

[30] MITRE Common Vulnerabilities and Exposures List. CVE-2001-0559. vixie cron vulnerability., 2001.

[31] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[32] Daniel S. Wilkerson and John D. Kubiatowicz. Hard Object: Hardware protection for software objects, 2008. United States Patent Application number 12/045,542, publication number US-2008-0222397-A1.

[33] Emmett Witchel. *Mondriaan Memory Protection*. PhD thesis, MIT, 2004.

[34] Emmett Witchel and Krste Asanović. Hardware works, software doesn't: Enforcing modularity with mondriaan memory protection. In *9th Workshop on Hot Topics in Operating Systems (HotOS-IX), Lihue, HI*, May 2003.

[35] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 304–316, New York, NY, USA, 2002. ACM Press.

[36] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *IEEE MICRO*, 2004.

[37] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *International Symposium on Computer Architecture ISCA*, 2004.

[38] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Simple and general architectural support for software debugging. *IEEE Top Picks in Computer Architecture*, 2004.

APPENDIX

A. ENSURING THE HARD OBJECT PROPERTIES

In Figure 6 we listed six computational properties that are guaranteed to programs running under a Hard Object system. In Section 3 we argued that these properties us achieve our goal of hardening object interfaces. In a Hard Object system these these six rather high-level properties are ensured by the enforcement by the hardware, verifier, system software, or the module's own compiler (not however requiring help from the compilers of other modules) of many very low-level *rules*. In this section we detail those rules.

- The hardware is trusted to follow the *H*-prefixed rules given in Figure 3.
- The runtime loader is augmented to include a *verifier* which checks that user-level code follows the *V*-prefixed rules before it is run.
- The compiler must generate code that will pass the verifier; additionally if the module it compiles is to be protected then the compiler-generated code must follow the *C*-prefixed rules.
- Modest modification is required to the libc and kernel memory management system such that they follow the *S*-prefixed rules.

We group the rules by subsection according to the property that they enforce; we start with an additional subsection for system-wide rules.

A.1 System-wide rules

S-INIT: Process initialization goes as follows. We allocate a stack per thread; the thread-specific framePointer and bottomOfStack set to top and bottom respectively. Stacks are disjoint from each other and the heap. The stack is owned by the empty range and stays that way throughout execution. All data pages are marked as non-text and all executable text pages have been through the verifier. Note that user-mode code cannot write to a text page.

V-CONTROL: The verifier checks that it can compute an over-approximation to the intra-procedural control flow graph and that this graph does not leave the function except at call and return sites. This requires jump and branch instructions to target fixed locations, unless they are part of the function call or return idiom. Computed gotos could also be allowed if sufficiently constrained, such as to a list of possibilities known at compile time – as is a C switch-statement. Note that a signal or a longjmp can still interrupt the runtime control flow; how this is handled is detailed below.

V-CONTROL-ATOMIC: Some verifier-enforced restrictions require that the program use a verifier-recognized idiomatic sequence of instructions; for example at a function return, after restoring the old framePointer the function must immediately return. Such sequences must be atomic for their properties to be static-time verifiable; thus the verifier checks that no branch or join node in the control flow graph occurs within any such idiomatic sequence. This analysis works even in multi-threaded code because we ensure threads do not access each other's stack frames.

A.2 Property P-ATOMIC

A function is called only at the top and does not return until it chooses to.

V-CALL: We wish to ensure that a function can only be called at its entry point. For direct function calls, it is easy to verify this while loading.

Calls through function pointers are handled in a manner similar to previous work on control-flow integrity [4]. We insert a 32-bit *callable* tag at the start of each function whose address is taken, and require that the program check for this tag before jumping through any function pointers. A suitable callable tag might be made using a no-op with a garbage argument.

V-CALLABLE-INTEGRITY: In an architecture allowing non-word-aligned instructions it is possible that the bit-string forming a callable tag might occur as a substring of a program, say spanning parts of more than one instruction; this must be prevented. The loader checks for this and if it occurs, the offending instruction combination is altered, perhaps by splitting it apart with an interposing no-op.

V-RETURN: We also wish to ensure that a function cannot be aborted abnormally. C++-style exceptions work fine since control must return to each stack frame temporarily to destruct any stack-allocated objects. We discuss longjmp() below.

A.3 Property P-FRAME

The stack frame of a function cannot be accessed by another function.

S-STACK-NOBODY: The operating system sets the pageOwner of stack pages to be the empty range. Therefore by H-ACCESS a stack page may only be accessed if it is between the framePointer and bottomOfStack registers.

C-STACK-HEAPIFY: Objects may no longer be allocated on the stack: if a pointer to such an object were passed to a function the framePointer would move at the function call and the object would be made inaccessible. Therefore all such objects must be allocated on the heap instead. Note that the auto-destruction semantics of C++ stack-allocated objects upon function return may still be implemented: the object is still semantically "on the stack" and the compiler ensures that the destructor is called at the right time.

V-FRAME: The verifier needs to check that the program uses the framePointer in such a way as to never expose any of the caller frames, including the saved framePointer and return pointer, except immediately before returning. For example, on a function call it is okay to

1. push the value of the framePointer, return address, and function arguments, and then
2. set the framePointer to point to the top of the arguments.

The old framePointer and return address are now protected for the duration of the call. For example, on a function return, it is okay to

1. move the framePointer up two words, exposing the old framePointer and return address,
2. immediately restore the old framePointer, and then
3. immediately jump to the return address.

C-FRAME-ZERO: If privacy is desired a function must zero its frame before returning.

Asynchronous control considerations

V-THREAD-MANAGER: When using user threads in order to allow thread context-switches the verifier must allow a trusted "thread context manager" to save and restore the framePointer and bottomOfStack registers of a thread in a thread control block in the context manager's reserved heap area.

S-SIGNAL: Unless the verifier can be sure that signal handlers ensure the same framePointer discipline as in *V-frame*, signal handlers must run on their own stack separate from any thread stack.

C-EXCEPTION: Recall that C++ style exceptions work fine since the exception unrolling must stop at each stack frame to destruct any "semantically" stack allocated objects (now on the heap); that is, they can follow the *V-return* frame discipline just as normal function return.

C-LONGJMP: Longjmp() is possible as follows. The setjmp/longjmp() module is trusted by the loader to manipulate the framePointer in a non-standard way (see below for the framePointer discipline). Functions that are prepared to allow a longjmp() through them without doing a proper return can annotate themselves as such. If the verifier can prove (possibly with help from programmer annotations to an augmented compiler) that only such functions may be jumped over, it allows the call to longjmp() to exist in a module; otherwise it does not.

A.4 Property P-KITH

A callee knows who its caller is.

V-KITH: On a cross-module function call, the program counter of the calling instruction must be passed as a *callerId* argument to the callee. Note that this value (or an offset of it, such as the address of the instruction following) is normally passed as part of the standard function call protocol so that a function knows to which address to return. The verifier needs to check that the correct value is passed so that the caller module may not fool the callee module as to the source of the function call.

A.5 Property P-KIN

An M-function can verify that a pointer refers to a properly-constructed M-object.

The primary OO idiom is for a module, or *class*, to construct upon request *objects* in its heap state, and to operate on them in well-defined ways at the request of other modules that may be untrusted. Such a module often takes a pointer, called 'this', to one of its previously-constructed objects as the argument to each function.

The P-KIN property ensures that each object reference passed from an untrusted source does indeed point to a valid object. We establish P-KIN from these sub-properties:

- P-KIN-PARTITION: Modules form a partition: their text ranges are mutually disjoint. Only whole modules own a page. Exactly one module owns a given page at any time.
- P-KIN-INTEGRITY: When the integrity bit is set on page the invariants of the owner module hold of the page.
- P-KIN-TYPE: An incoming 'this' pointer points to an object of the correct type.

Basic memory allocation considerations

S-BRK: Any system calls that map memory into the virtual address space, such a brk() or shmget(), take an additional argument, Range *callerOwnerRange*, as the pageOwner of new pages.

S-MALLOC: Similarly, user-level memory allocators, such as malloc(), must take the same argument if they are to function properly. Two levels of ownership transfers are necessary: first malloc() asks brk() to make a (probably large) block of memory and set its owner to malloc(); later malloc() uses set_pageOwner to transfer ownership of smaller blocks to client code. However, correctness does not rely on malloc behaving this way: if malloc misbehaves a module may not get any memory back, but it will not be corrupted.

C-ALLOC-LOCAL: The client must request whole pages from malloc as that is the granularity of ownership. To implement traditional byte-granularity allocation, the traditional call to malloc() might be changed to a call to a module-local allocator that further partitions the pages for the client.

Sub-property P-KIN-PARTITION

V-PARTITION: The verifier computes the text range of a module as it is loaded. The verifier checks that the ranges do not overlap. Recall that the compiler cannot necessarily compute module ranges for different translation units so a module must specify other modules by name; these names are replaced by the loader at load time. If an argument to set_pageOwner is a static module name then loader replaces it with the literal text range of that module.

V-PARTITION-DYNAMIC: Any module name used as a runtime value is replaced with an index into the verifier's table of module text ranges. If an argument to set_pageOwner is a runtime module identifier then the loader inserts a check just before the instruction. The check is a switch statement: a lookup into a table embedded into the text together with a range check against an embedded table size ensuring that the module id is a legal index into the table. The result of the lookup is used as the argument to set_pageOwner.

Sub-property P-KIN-INTEGRITY

C-INTEGRITY: A module C must be able to verify that it can trust that the data on its page satisfies its invariants; otherwise an untrusted module U could create a subtly-corrupted page, transfer its ownership to C, and call a method of C passing it.

Recall that H-OWNER guarantees that whenever ownership of a page is transferred the integrity bit is cleared. The compiler generates code so that a module receiving a page with no integrity bit first ensures the page satisfies its invariants before setting the integrity; most commonly this would occur when getting a page back from malloc(). Methods expecting a 'this' pointer can simply refuse to operate if the page 'this' points to does not have the integrity bit set.

Sub-property P-KIN-TYPE

C-TYPE: When a module owns more than one type of object, it must check that any pointer it gets from an untrusted source points to an object of the expected type. Otherwise, malicious code could trick a module into operating on an object of the wrong type.

The easiest way for a compiler to enforce this is by placing different types of objects on different pages. By marking the start of each page with a *type identifier*, the compiler can insert checks that a pointer refers to a page holding only objects of the expected type. The integrity of this identifier is one of the page invariants that P-KIN-INTEGRITY enforces.

Additionally, a module must verify that it is not passed a pointer into the middle of an object. The compiler can easily insert code to subtract and compute a modulus in order to check that the pointer is aligned with the start of the page with respect to the object size. The object size can be hard-coded if known at compile time, or stored in the page header if not. Another solution for checking object alignment is to use a "pointers to pointers" page layout: each page has a contiguous block of pointers at the top that point to the actual objects on the same page. Situations where the object size is not know at compile time can occur in situations that arise due to inheritance; see Section 4.

A.6 Property P-OBJECT

An M-object cannot be accessed by a function not in M.

The H-ACCESS rule guarantees that only the owner can touch the pages.

A.7 Property P-FREE

Only M can destroy an M-object.

S-FREE: Recall that H-OWNER prevents any other module from taking the ownership from a page. The brk() system call refuses to unmap pages that are still owned by a user module unless that module is the one making the call to brk(). To free a page, a module must transfer its ownership to malloc()/free(), then call free(); free then may call brk().