Evaluating Hard Object: a Lightweight Module Isolation System

Kevin Klues, Derrick Coetzee

In Collaboration with: Daniel Wilkerson, Mark Winterrowd, and John Kubiatowicz

Final Project
CS261 - Computer Security
University of California Berkeley
Berkeley, CA

{klueska, dcoetzee}@eecs.berkeley.edu
{daniel.wilkerson, markwinterrowd4}@gmail.com, kubitron@cs.berkeley.edu

ABSTRACT

Because large, complex systems inevitably contain security vulnerabilities, it is important to mitigate their impact. One of the most effective methods for doing so is *privilege separation*, the separation of an application into *modules* such that the compromise of one module grants the attacker only the limited privileges possessed by that module. However, the traditional hardware address space mechanism used to isolate processes exhibits high communication overhead between modules, making it impractical for fine-grained privilege separation. Software-based fault isolation (SFI) largely eliminates communication overhead, but provides less effective isolation and imposes substantial complexity and runtime overhead.

Hard Object, a hybrid hardware/software solution, achieves the best of both worlds by providing a model similar to SFI but performing the most expensive operations in simple hardware extensions. The system, designed as an extension of a 32-bit x86 PC platform, is language-agnostic, and provides strong guarantees suitable for isolating malicious code.

In this report, we present details of an (incomplete) reference implemention of Hard Object based on the cycle-accurate 32-bit x86 simulator PTLSim[11]; a companion document that accompanies this submission presents the full details of the design. Additionally, we provide an evaluation of a modularized `passwd` application running on our modified simulator. For this application, we see a slowdown of 2.6% as compared to a reference implementation running on the unmodified simulator. In the future we plan to complete the simulator and evaluate a more comprehensive set of applications.

1. INTRODUCTION

Large software products involve the interaction of a large number of complex components totalling in the millions of lines of code, and are often written by multiple parties with varying degrees of code quality. An example is a web browser, which must implement network protocols, parse a variety of complex data formats, execute scripts, perform layout and rendering, and integrate with third-party plug-ins. Strong verification techniques such as model checking do not scale to systems of this size and complexity. Systems of this type are a rich source of undiscovered security vulnerabilities.

Privilege separation is an established technique for mitigating the impact of undiscovered security vulnerabilities. An application is divided into *modules*, and each module is, according to the *principle of least privilege*, only permitted to perform operations that it requires to complete its task. If a module contains a security vulnerability, it will, at worst, lead to the attacker gaining complete control of that module and acquiring its limited privileges. In practice, large applications can often be factored such that most of the program requires minimal privileges to run; security verification and auditing effort can then be focused on the small part of the program running with elevated privilege. A compelling example is given by Provos et al's privilege separation of the OpenSSH server in 2003, which uses a system call runtime trace utility to facilitate the creation of policies.[6]

Software engineering best practice already dictates that large applications should be factored into loosely coupled modules. However, particularly in non-memory-safe languages such as C and C++, these module boundaries are not enforced, and a module which is compromised can often assert direct control over the entire system. Privilege separation depends on a strong isolation mechanism which can guarantee, among other things, that modules do not corrupt the memory of other modules, and that they communicate in a controlled manner.

The most well-established system for module isolation is hardware address spaces, which allow each module to have its own view of memory; if a module cannot even address the memory of other modules, it can't corrupt it. Unfortunately, address spaces make communication between modules expensive, primarily because control transfer between modules necessarily involves a context switch. Wahbe et al[8] estimate that a context switch requires on the order of 100 times longer than a simple function call. Because of this, address spaces are only used in practice for coarse-grained, rarely-communicating modules, leading to lost opportunities for privilege separation.

Software-based fault isolation (SFI), introduced by Wahbe et al,[8] presents an alternative model in which all modules run together in a single address space, but each module is restricted to being able to access only a subset of addresses. Since the current module can be easily changed at any time, communication is again as cheap as a simple function call. However, enforcing the access restrictions using purely software-based mechanisms proves complex and expensive; every read and write must be instrumented to perform runtime checks, and control flow integrity must be enforced to prevent these checks from being circumvented. To keep these checks fast, efficient SFI systems typically protect only writes, and provide limited flexibility in how addresses can be assigned to modules. This flexibility is inadequate to support idioms like the program stack, which contains data belonging to many different modules.

The essential insight of Hard Object is that the key to making the SFI model practical is to provide simple hardware extensions that perform checks that are too expensive to implement in software. On every read and write, the hardware enforces that the memory accessed belongs to the current module, which is in turn determined by the program counter. To support typical programs, Hard Object provides three distinct types of protection:

1. **Heap/global protection**: Each code or data page is assigned a module ID in the page table. Code can only read or write data with the same module ID.
2. **Stack protection**: Two special registers, the caller-protect register and the bottom of stack register, delineate the accessible region of the stack. When calling another module, the caller can protect its stack by updating the caller-protect register. A separate register, the flow-protect register, is used to ensure the call returns normally by protecting the return address.
3. **Call protection**: If modules can be entered at any point, malicious code can run segments of them to achieve privilege escalation, as in return-oriented programming.[7] Each module marks the points where control is allowed to enter the module using a special instruction, and the hardware enforces this.

To support privilege separation, Hard Object also allows filtering of system calls by module. In the simplest case, only a single module is allowed to make system calls, allowing it to interpose on all access to system services based on the module ID of the caller.

Hard Object was first described in a technical report published in July 2009,[9] describing a preliminary design and a preliminary evaluation based on simple performance estimates. The present work describes progress on a more ambitious evaluation incorporating a hardware simulator for the proposed system, a fully privilege-separated application designed for the Hard Object platform, and an improved design.

Section 2 describes

2. RELATED WORK

A number of systems have attempted to provide lightweight module isolation on top of existing commodity hardware. Typically these use some combination of software and creative use of some existing hardware mechanism such as segmentation or address spaces; they are each specific to a particular architecture. To instrument code, each incorporates either a trusted compiler/verifier, or dynamic binary rewriting.

The first software-based fault isolation (SFI) system was described in 1993 by Wahbe et al.[8] It ran on DEC systems based on MIPS and Alpha RISC instruction sets. It assigns a contiguous range of addresses with a common bit prefix called a *segment* to each module, for example addresses `0x1f000000` through `0x1fffffff`. Using a trusted compiler it maintains the invariant that the *dedicated registers* always point into the current module's segment, and that all writes and indirect jumps are made through the dedicated registers. The system is not purely software-based – it relies on address spaces to eliminate some checks. All cross-module calls must pass through special call and return stubs, and each module has a private stack. It exhibits roughly 4% runtime overhead when checking writes only and 20% overhead when checking reads and writes.

While simple and reasonably efficient, Wahbe et al's scheme is specialized to RISC architectures: it assumes five reserved registers are available, that there are no complex addressing modes, and that jumps do not target the middle of instructions. Cross-module calls are about 10 times slower than ordinary function calls, due to the overhead of the call and return stubs, and unlike in Hard Object stack-based arguments must be copied. It also assigns a single region to each module, whereas Hard Object allows ownership to be specified at page granularity. Finally, the use of a trusted compiler results in a large trusted code base; compiler bugs could lead to security vulnerabilities in its output.

In 2006, the MIT's PittSFIeld system[5] extended SFI to CISC architectures, with a runtime overhead of about 21% when protecting writes only. The correctness of their design is established by a machine-checkable proof, and they avoid trusting the compiler by having a trusted verifier at load-time. It overcomes the issue of unaligned instructions by grouping instructions into 16-byte chunks of straight line code and constraining jumps to target the beginning of chunks. By ensuring that all accessed are checked within the same chunk, there is no need for dedicated registers.

Vx32 (2008) is another SFI system for x86 that achieves lower overhead than PittSFIeld by protecting data accesses using the x86's legacy segmentation support, and by protecting control flow using dynamic binary rewriting.[3] Untrusted modules cannot invoke one another, and invoking the trusted module requires trapping out of the binary rewriter. The deprecation of segmentation for native code on 64-bit platforms makes Vx32 an unlikely option for the future – if a hardware mechanisms is to replace it, it should be one designed for module isolation.

As opposed to the above systems, which each use a single segment per module, Microsoft Research's BGI (Byte-Granularity Isolation) system [2] for x86 platforms allows module ownership of addresses to be specified at byte granularity, even finer than Hard Object , allowing correct module ownership of the stack to be specified directly. Although runtime overhead is good, 6.4% on average, there is a memory overhead of 12% of all data. In order to keep this overhead reasonable, strong restrictions were imposed on the number of modules (no more than about 20) and on the number of distinct type IDs that can be assigned to words. BGI also relies on a trusted compiler.

An alternative to SFI is the use of strongly-typed languages; these were used for example to isolate modules in the Singularity microkernel.[4] While this approach can be effective in eliminating performance overhead, it depends on a trusted code base, usually including some type of compiler and a runtime engine, which are themselves large enough to be vulnerable to security issues. It is also of little assistance to legacy, performance-critical, and resource-constrained applications.

Mondriaan Memory Protection[10] (MMP) is a pure hardware-based module isolation system, which similarly to BGI allows word-granularity ownership – with the same problem, high memory overhead. Sophisticated hardware compression reduces this to 1% of all data for coarse-grained modules, or 9% for fine-grained modules; but at the expense of substantial hardware complexity. Segments can be mapped into multiple locations with different privileges, which is convenient for sharing data between modules and more flexible than Hard Object's module ID prefix scheme for data sharing. Certain programs unpredictably cause a dramatic increase in both space (44%) and time (100%) overhead.

As compared to other systems, Hard Object is designed to enable page-granularity isolation, to support a large number of modules, and to make cross-module calls as fast as possible, comparable to a standard function call, with less costly hardware modifications than Mondriaan. However, our performance results indicate that cross-module calls are presently quite expensive, requiring an additional 500 cycles per call, even in the best case where all registers are caller-save; these are intended to eventually be implemented in hardware, but they may still remain expensive.

3. HARD OBJECT OVERVIEW

As mentioned in Section 1, the full Hard Object design contains mechanisms to protect 1) per module heap allocated and global variables, 2) per module stack frames, and 3) cross module calls to public functions. While the companion document submitted with this report goes into all of the details on each of these mechanisms and how they work, we provide here a basic overview of the mechanisms important for understanding the implementation described in Section 4 and the evaluation presented in Section 6.

caller locals ···
old caller-protect
old flow-protect ← caller-protect
callee arguments ···
return address ← flow-protect
callee locals ··· ← stack-pointer
··· ← bottom-of-stack

Identify operating modules:

- **caller-module-id**: prev current-module-id
- **current-module-id**: owner-module-id of current text page

Ensure stack integrity:

- **caller-protect**: location of bottom of caller locals of last protected call

- **flow-protect**: location of saved return address of last protected call
- **bottom-of-stack**: max extent of the stack

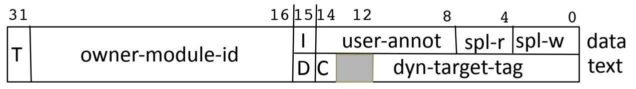

Figure 1: The page table extensions required by Hard Object .

4. IMPLEMENTATION

We have integrated a reference implementation of Hard Object into a cycle accurate 32-bit x86 simulator called PTLSim []. As shown in Figure 2, PTLSim operates by spawning off a child process of the program it wishes to simulate, injecting itelf into that child process's address space, adjusting the entry point of the child process to point to the entry point of the PTLSim runtime, and then launching the program inside this runtime. PTLSim makes heavy use of the non-posix system call `ptrace` in order to accomplish this task.

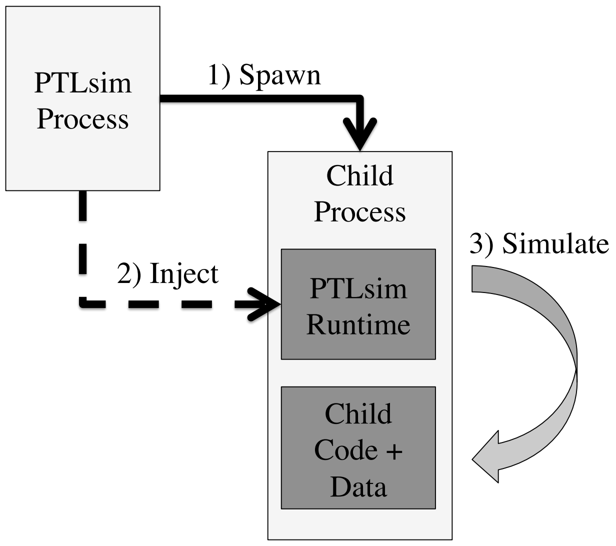

Figure 2: The PTLsim simulator. It operaties by spawning off a child process of the program it is going to simulate, injecting its image into the child process's address space, adjusting the entry point of the program to that of the PTLSim runtime, and then simulating the program.

We modified PTLSim to include 1) the ability to maintain extended page table entries for each page allocated to simulated program 2) the ability to load a binary modularized for Hard Object and set up its extended page table entries correctly, 3) instructions for getting/setting all of our new Hard Object registers, and 4) a subset of the checks used to maintain heap and stack integrity on a per module basis. Additionally, we leverage PTLSims ability to interpose on system calls to intercept all calls that allocate memory and set up their extended page table entries appropriately. We leave it as future work to integrate the rest of the Hard Object design into the simulator.

4.1 The Hard Object Loader

In order to develop a Hard Object loader for PTLSim, we needed a way of allowing application developers to encode how their applications had been modularized, as well as announce what their module ids were. To facilitate this, we wrote a simple scripting language that allows developers to declare which C source files to associate with a module and what they want their module ids to be. This script is then parsed and fed to a tool we wrote which intreposes on the linking of the final program executable. This tools job is to maintain separate ELF sections for each of the sections found in object files associated with different modules. Each section name is prepended with information pertinent to a given module, so that it can later be parsed and different sections from different modules can be distinguished. The Hard Object loader built in to PTLSim simply knows how to parse one of these modified ELF files and set up the page table entries for each section in each module appropriately.

4.2 Getter/Setter Instructions

PTLSim translates all x86 instructions that it sees in an execution stream into a set of micro-operations (uops) before running them through a simulated out-of-order core. In order to maintain the cycle-accurate properties of PTLSim, we implemented our Hard Object getter/setter instructions using these same set of uops that regular instructions are broken down into. We then check to make sure that we haven't broken any of the invariants that must hold for the values these registers can be set to, and issue a fault if something goes wrong.

5. EXAMPLE APPLICATION

The command-line UNIX utility `passwd` allows users to change their password and the UNIX superuser to change any user's password. Because it needs to write the `/etc/passwd` file, which is only writable by the superuser, it must execute with superuser privileges; consequently, any security vulnerability in this application could be used to compromise the system.

As a demonstration of privilege separation in Hard Object , we implemented a modularized version of `passwd` which runs under our partially-implemented hardware simulator. Including blank lines and comments, the entire application consists of 578 lines of C code. Although source-to-source translation tools for facilitating large-scale application migration are under development, these tools were not used to build this simple demonstration application.

5.1 Application Modules

As seen in Figure 4, the `passwd` application is divided into five modules: `master`, `main`, `parseui`, `readpasswd`, and `updatepasswd`. A description of each module and its responsibilities is detailed below; the public interfaces exposed by the modules are listed in Figure 3.

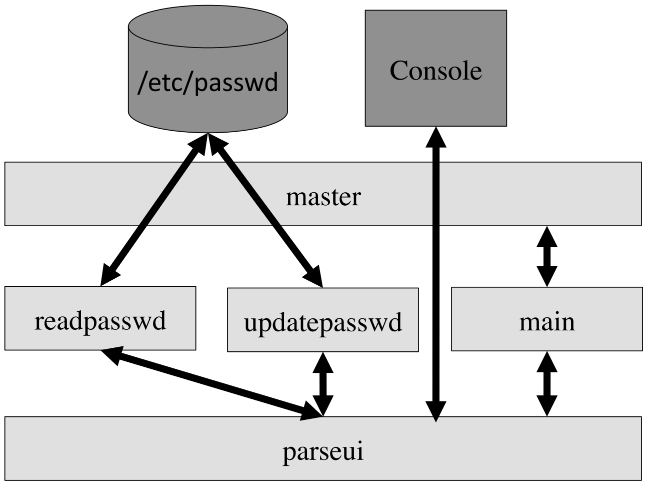

Figure 4: The `passwd` application modularized for Hard Object

master: The `master` module is the only module permitted to make system calls (other than allocating memory) and interposes on all system calls made by other modules, using the caller module ID to check that the calls are permitted. It supports a limited set of operations, many of which take advantage of publically-readable heap pages to pass pointers to read-only strings to other modules, as shown in Figure 3.

main: Contains the entry point `main()`. Because `passwd` depends on the current user ID, but the simulator only runs as root, main simulates running as another user by taking the desired uid on the command line and calling `master_setuid()`. This module also launches the `parserui` module and performs the final exit. This module contains no publically callable functions.

parserui: Handles the UI, writing prompts and reading old and new passwords. Could also, in theory, do password complexity validation. This module contains a single publically callable function as seen below.

readpasswd: Implements utilities that read the passwd file but don't write it, except for password checking. This module contains a single publically callable function as seen below.

updatepasswd: Implements core password checking and updating functionality. A description of the interface provided by this module can be seen below, along with a description of what each function and what rights each user has when calling it.

The primary security goal achieved by this separation of privilege is that if the `parserui` module is compromised, there is no loss of security – it is not possible to perform any operations through the other module interfaces that cannot already be done through the normal console UI. Additionally, compromising the `readpasswd` module can only expose the contents of the passwd file, not allow writing to it.

5.2 Issues

One problem that arose while designing the applciation's privilege separation was whether to place the is_correct_password() function in the `updatepasswd` or the `readpasswd` module. Although it does not require write access to the passwd file itself, it is called directly by the `update_password()` function in the `updatepasswd` module. If it were located in the `readpasswd` module and that module were to become compromised, it would allow a user to bypass the password check in `update_password()` and set the root password, resulting in a complete system compromise.

A difficult problem was how to avoid exposing password hashes to the `parserui` module. Because the file contents of `/etc/passwd` are shared on a publically-readable page, any running module can view them. Ideally only `readpasswd`, `updatepasswd`, and `master` would be able to read these pages, but this type of limited sharing was not yet implemented. Instead, we opted to simply have each call to the `readpasswd` or `updatepasswd` module re-read the passwd from the disk, and memset it to zero when it's done (via `master_done_with_passwd_file_contents()`). Although this seems to work, there may be problems with libc buffering a copy of the data in publically readable legacy data pages.

Because the hardware support for cross-module call and return instructions has not yet been fully integrated into PTLSim, the required functionality, which primarily involves manipulating Hard Object special registers, was simulated in software during the development of this application (which makes the present system insecure and only suitable for benchmarks). Below is a list of the functionality that needed to be simulated. Certain functionality is duplicated across modules; for example, each module receives its own allocator and string concatenation function. This allows each module's copy to operate on private data without using complex ownership transfer mechanisms, at the cost of greater code size.

6. EVALUATION

Figure 5 compares two runs of the modularized *passwd*

Module	Function	Callable by	Description
master	master_srand	main	Seeds the random number generator with the current time.
master	master_print	all	Prints a string to the console.
master	master_read_passwd_file_contents	readpasswd, updatepasswd	Reads the contents of the passwd file to a buffer and returns a pointer to it.
master	master_done_with_passwd_file_contents	readpasswd, updatepasswd	Erases the contents of the passwd file contents buffer
master	master_write_passwd_file_contents	updatepasswd	Replaces the passwd file with the given string.
master	master_getuid	all	Gets the current user ID.
master	master_setuid	main	Sets the current user ID. *(Workaround for simulator limitation)*
master	master_abort	all	Prints a message to stderr and aborts the program.
master	master_exit	main	Exits the program with the given error code.
master	master_input_line	all	Inputs a line from the console to a buffer and returns a pointer to it.
parserui	process_command	main	Given a command line, runs the application UI.
readpasswd	get_current_user_name	all	Gets the current username by looking up the current user ID in the passwd file.
updatepasswd	is_correct_password	all	Checks for a given user if the given password is correct. Only root may specify a user.
updatepasswd	update_password	all	Updates a user's password. Only root may specify a user; only root may set the current password to NULL.

Figure 3: Public interfaces exposed by the `passwd` modules.

	Instructions	Cycles
Without Hard Object	477611	632634
With Hard Object	479943	648961
Overhead	2332 (0.5%)	16327 (2.6%)
Overhead per call	66.63	466.5

Figure 5: Performance measurements of a run of passwd with and without Hard Object checks, and averaged over the 35 cross-module calls.

application with and without Hard Object features enabled. Overhead reflects costs such as updating the stack and module ID registers on each cross-module call, and performing caller module ID checks. Because most of the CPU time in *passwd* is spent in *crypt()* computing the password hash, a trivial *crypt()* was substituted to emphasise the modularized portion of the program.

An important point to evaluate was that injected faults would be detected. Write checks to the stack and heap have not yet been implemented. The limited stack checks implemented in software passed correctly. However, both system call filtering and caller module ID checks are fully implemented, allowing simple sandboxing of system calls to be verified. During the run, no modules other than the master module perform system calls, and if the parser_ui module is instrumented to attempt to write to the password file through the master module's interface, it fails with the desired error ("Module other than update_passwd module tried to write to passwd file").

7. DISCUSSION AND FUTURE WORK

A number of the features proposed by the Hard Object design in order to support idiomatic programming were validated by our example application. These include the use of publically-readable heap pages (convenient for sharing string data read-only); the ability to limit syscalls to a single "gateway" module; and the ability to invoke legacy code, in this case libc, not compiled for use with Hard Object .

The example application also points to some important limitations in the existing design: legacy code frequently passes pointers to its own data pages back to application code, requiring them to be readable, but also internally stores private buffered data, such as the password hashes read from `/etc/passwd`. Determining a privilege policy for legacy code is a difficult problem. The tools also provide little assistance in choosing a good privilege separation, a task that currently requires careful thoght and knowledge of the problem domain, or in easily modifying a separation at a later time.

The most important future work is completion of the hardware simulator. In particular, the hardware simulator does not currently perform write checks on the heap or stack, which are essential for verifying that test applications do not perform illegal writes. Although the example application is written to pass the write checks implemented by Hard Object , programming errors could still result in runtime failures. Additional hardware simulator work includes hardware implementation of cross-module call and return instructions (currently done in application software), and the implementation of the shadow stack (for legacy tool support). TLB fields need to be made accessible so the integrity bit can be set, and the integrity bit must be cleared at the right time and checked during writes. Callbacks from legacy code require special support that has not been implemented.

On the application side, evaluation needs to be extended to large-scale existing applications, such as browsers, web servers, or microkernels. Determining a good privilege separation and modularizing large applications is a complex task, particularly considering the special requirements of Hard Object modules – for example, addresses cannot be taken of stack-allocated data, and heap-allocated objects cannot be used as in/out parameters without an explicit page ownership transfer. The source-to-source transformation tools, still under construction, are intended to facilitate some of these migration steps; we may also benefit from existing automatic privilege separation tools such as Privtrans[1]. Cross-cutting concerns like exception handling and garbage collection have not been dealt with.

Beyond the initial Hard Object design, there is important work in generalizing the platform. The 32-bit machines it is designed for are being phased out, and the design will have to be updated for 64-bit x86 machines, as well as RISC platforms such as the ARM instruction sets used on mobile devices. Additionally, page-granularity layout imposes substantial software complexity upon the mechanisms for allocation and transferring objects between modules; future designs could incorporate more flexible fine-grained address ownership assignment in order to simplify these scenarios. Similarly, the inability to pass arguments on the stack cross-module complicates existing programs and it would be useful to avoid this if possible. Another useful feature would be a means for sharing identical code between modules, to decrease code size while maintaining the benefit of having logical private copies that can operate on private data.

8. ACKNOWLEDGEMENTS

This work has been done in collaboration with Daniel Wilkerson, Mark Winterrowd and John Kubiatowicz. All members of this project contributed equally to the overall design of the system, which went through many iterations before settling on the one presented in this paper. Other than for the design itself, the various tasks undertaken during this project can be fairly easily at-

tributed to individual people. Daniel is responsible for overseeing the project and maintaining the design documents, as well as for implementing static analyses and source-to-source transformation tools intended to assist in application migration; while not used in our final evaluation, these will be important for later publications. Kevin is responsible for adding all of the Hard Object registers, instructions, invariant checks, and module loader into the PTLSim simulator, as well as for designing and implementing the framework for specifying how to group different C source files into modules, specify their properties, and build them in a format recognizable by the PTLSim loader. Mark is responsible for helping Kevin figure out how the internals of PTLSim worked, as well as helping to add various registers and instructions into PTLSim itself. Derrick is responsible for partially implementing a C compiler backend during the early stages of our design (which we never ended up using), for work on the source-to-source transformation tools, and for implementing the modularized `passwd` application used in our final evaluation. Finally, John Kubiatowicz is responsible for helping to design and evaluate the feasibility of the hardware component of Hard Object.

9. REFERENCES

[1] D. Brumley and D. Song. Privtrans: automatically partitioning programs for privilege separation. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[2] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58, New York, NY, USA, 2009. ACM.

[3] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

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

[5] S. McCamant and G. Morrisett. Evaluating sfi for a cisc architecture. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[6] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[7] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, New York, NY, USA, 2007. ACM.

[8] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, New York, NY, USA, 1993. ACM.

[9] D. Wilkerson, D. A. Molnar, M. Harren, and J. D. Kubiatowicz. Hard-object: Enforcing object interfaces using code-range data protection. Technical Report UCB/EECS-2009-97, EECS Department, University of California, Berkeley, Jul 2009.

[10] E. Witchel, J. Cates, and K. 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.

[11] M. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:23–34, 2007.