Code-Pointer Integrity

Volodymyr Kuznetsov*, László Szekeres‡, Mathias Payer†,§
George Candea*, R. Sekar‡, Dawn Song†

*École Polytechnique Fédérale de Lausanne (EPFL),
†UC Berkeley, ‡Stony Brook University, §Purdue University

Abstract

Systems code is often written in low-level languages like C/C++, which offer many benefits but also delegate memory management to programmers. This invites memory safety bugs that attackers can exploit to divert control flow and compromise the system. Deployed defense mechanisms (e.g., ASLR, DEP) are incomplete, and stronger defense mechanisms (e.g., CFI) often have high overhead and limited guarantees [19, 15, 9].

We introduce code-pointer integrity (CPI), a new design point that *guarantees the integrity of all code pointers* in a program (e.g., function pointers, saved return addresses) and thereby prevents all control-flow hijack attacks, including return-oriented programming. We also introduce code-pointer separation (CPS), a relaxation of CPI with better performance properties. CPI and CPS offer substantially better security-to-overhead ratios than the state of the art, they are practical (we protect a complete FreeBSD system and over 100 packages like apache and postgresql), effective (prevent all attacks in the RIPE benchmark), and efficient: on SPEC CPU2006, CPS averages 1.2% overhead for C and 1.9% for C/C++, while CPI's overhead is 2.9% for C and 8.4% for C/C++.

A prototype implementation of CPI and CPS can be obtained from `http://levee.epfl.ch`.

1 Introduction

Systems code is often written in memory-unsafe languages; this makes it prone to memory errors that are the primary attack vector to subvert systems. Attackers exploit bugs, such as buffer overflows and use after free errors, to cause memory corruption that enables them to steal sensitive data or execute code that gives them control over a remote system [44, 37, 12, 8].

Our goal is to secure systems code against all *control-flow hijack* attacks, which is how attackers gain remote control of victim systems. Low-level languages like C/C++ offer many benefits to system programmers, and we want to make these languages safe to use while preserving their benefits, not the least of which is performance. Before expecting any security guarantees from systems we must first secure their building blocks.

There exist a few protection mechanism that can reduce the risk of control-flow hijack attacks without imposing undue overheads. Data Execution Prevention (DEP) [48] uses memory page protection to prevent the introduction of new executable code into a running application. Unfortunately, DEP is defeated by code reuse attacks, such as return-to-libc [37] and return oriented programming (ROP) [44, 8], which can construct arbitrary Turing-complete computations by chaining together existing code fragments of the original application. Address Space Layout Randomization (ASLR) [40] places code and data segments at random addresses, making it harder for attackers to reuse existing code for execution. Alas, ASLR is defeated by pointer leaks, side channels attacks [22], and just-in-time code reuse attacks [45]. Finally, stack cookies [14] protect return addresses on the stack, but only against continuous buffer overflows.

Many defenses can improve upon these shortcomings but have not seen wide adoption because of the overheads they impose. According to a recent survey [46], these solutions are incomplete and bypassable via sophisticated attacks and/or require source code modifications and/or incur high performance overhead. These approaches typically employ language modifications [25, 36], compiler modifications [13, 3, 17, 34, 43], or rewrite machine code binaries [38, 54, 53]. Control-flow integrity protection (CFI) [1, 29, 53, 54, 39], a widely studied technique for practical protection against control-flow hijack attacks, was recently demonstrated to be ineffective [19, 15, 9].

Existing techniques cannot both *guarantee protection* against control-flow hijacks and impose *low overhead* and *no changes* to how the programmer writes code. For example, memory-safe languages guarantee that a memory object can only be accessed using pointers properly based on that specific object, which in turn makes control-flow hijacks impossible, but this approach requires runtime checks to verify the temporal and spatial correctness of pointer computations, which inevitably induces undue overhead, especially when retrofitted to memory-unsafe languages. For example, state-of-the-art memory safety implementations for C/C+ incur $\geq 2\times$

Appears in Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2014

overhead [35]. We observe that, in order to render control-flow hijacks impossible, it is sufficient to guarantee the integrity of code pointers, i.e., those that are used to determine the targets of indirect control-flow transfers (indirect calls, indirect jumps, or returns).

This paper introduces code-pointer integrity (CPI), a way to enforce precise, deterministic memory safety for all code pointers in a program. The key idea is to split process memory into a *safe region* and a *regular region*. CPI uses static analysis to identify the set of memory objects that must be protected in order to guarantee memory safety for code pointers. This set includes all memory objects that contain code pointers and all data pointers used to access code pointers indirectly. All objects in the set are then stored in the safe region, and the region is isolated from the rest of the address space (e.g., via hardware protection). The safe region can only be accessed via memory operations that are proven at compile time to be safe or that are safety-checked at runtime. The regular region is just like normal process memory: it can be accessed without runtime checks and, thus, with no overhead. In typical programs, the accesses to the safe region represent only a small fraction of all memory accesses (6.5% of all pointer operations in SPEC CPU2006 need protection). Existing memory safety techniques cannot efficiently protect only a subset of memory objects in a program, rather they require instrumenting *all* potentially dangerous pointer operations.

CPI fully protects the program against all control-flow hijack attacks that exploit program memory bugs. CPI requires no changes to how programmers write code, since it automatically instruments pointer accesses at compile time. CPI achieves low overhead by selectively instrumenting only those pointer accesses that are necessary and sufficient to formally guarantee the integrity of all code pointers. The CPI approach can also be used for data, e.g., to selectively protect sensitive information like the process UIDs in a kernel.

We also introduce code-pointer separation (CPS), a relaxed variant of CPI that is better suited for code with abundant virtual function pointers. In CPS, all code pointers are placed in the safe region, but pointers used to access code pointers indirectly are left in the regular region (such as pointers to C++ objects that contain virtual functions). Unlike CPI, CPS may allow certain control-flow hijack attacks, but it still offers stronger guarantees than CFI and incurs negligible overhead.

Our experimental evaluation shows that our proposed approach imposes sufficiently low overhead to be deployable in production. For example, CPS incurs an average overhead of 1.2% on the C programs in SPEC CPU2006 and 1.9% for all C/C++ programs. CPI incurs on average 2.9% overhead for the C programs and 8.4% across all C/C++ SPEC CPU2006 programs. CPI and CPS are effective: they prevent 100% of the attacks in the RIPE benchmark and the recent attacks [19, 15, 9] that bypass CFI, ASLR, DEP, and all other Microsoft Windows protections. We compile and run with CPI/CPS a complete FreeBSD distribution along with ≥ 100 widely used packages, demonstrating that the approach is practical. This paper makes the following contributions:

1. Definition of two new program properties that offer a security-benefit to enforcement-cost ratio superior to the state of the art: code-pointer integrity (CPI) guarantees control flow cannot be hijacked via memory bugs, and code-pointer separation (CPS) provides stronger security guarantees than control-flow integrity but at negligible cost.
2. An efficient compiler-based implementation of CPI and CPS for unmodified C/C++ code.
3. The first practical and complete OS distribution (based on FreeBSD) with full protection built-in against control-flow hijack attacks.

In the rest of the paper we introduce our threat model (§2), describe CPI and CPS (§3), present our implementation (§4), evaluate our approach (§5), discuss related work (§6), and conclude (§7). We formalize the CPI enforcement mechanism and provide a sketch of its correctness proof in Appendix A.

2 Threat Model

This paper is concerned solely with control-flow hijack attacks, namely ones that give the attacker control of the instruction pointer. The purpose of this type of attack is to divert control flow to a location that would not otherwise be reachable in that same context, had the program not been compromised. Examples of such attacks include forcing a program to jump (i) to a location where the attacker injected shell code, (ii) to the start of a chain of return-oriented program fragments ("gadgets"), or (iii) to a function that performs an undesirable action in the given context, such as calling `system()` with attacker-supplied arguments. Data-only attacks, i.e., that modify or leak unprotected non-control data, are out of scope.

We assume powerful yet realistic attacker capabilities: full control over process memory, but no ability to modify the code segment. Attackers can carry out arbitrary memory reads and writes by exploiting input-controlled memory corruption errors in the program. They cannot modify the code segment, because the corresponding pages are marked read-executable and not writable, and they cannot control the program loading process. These assumptions ensure the integrity of the original program code instrumented at compile time, and enable the program loader to safely set up the isolation between the safe and regular memory regions. Our assumptions are consistent with prior work in this area.

3 Design

We now present the terminology used to describe our design, then define the code-pointer integrity property (§3.1), describe the corresponding enforcement mechanism (§3.2), and define a relaxed version that trades some security guarantees for performance (§3.3). We further formalize the CPI enforcement mechanism and sketch its correctness proof in Appendix A.

We say a pointer dereference is *safe* iff the memory it accesses lies within the target object on which the dereferenced pointer is based. A *target object* can either be a memory object or a control flow destination. By *pointer dereference* we mean accessing the memory targeted by the pointer, either to read/write it (for data pointers) or to transfer control flow to its location (for code pointers). A *memory object* is a language-specific unit of memory allocation, such as a global or local variable, a dynamically allocated memory block, or a sub-object of a larger memory object (e.g., a field in a struct). Memory objects can also be program-specific, e.g., when using custom memory allocators. A *control flow destination* is a location in the code, such as the start of a function or a return location. A target object always has a well defined lifetime; for example, freeing an array and allocating a new one with the same address creates a different object.

We say a pointer is *based on* a target object X iff the pointer is obtained at runtime by (i) allocating X on the heap, (ii) explicitly taking the address of X, if X is allocated statically, such as a local or global variable, or is a control flow target (including return locations, whose addresses are implicitly taken and stored on the stack when calling a function), (iii) taking the address of a sub-object y of X (e.g., a field in the X struct), or (iv) computing a pointer expression (e.g., pointer arithmetic, array indexing, or simply copying a pointer) involving operands that are either themselves based on object X or are not pointers. This is slightly stricter version of C99's "based on" definition: we ensure that each pointer is based on at most one object.

The execution of a program is *memory-safe* iff all pointer dereferences in the execution are safe. A program is memory-safe iff all its possible executions (for all inputs) are memory-safe. This definition is consistent with the state of the art for C/C++, such as SoftBounds+CETS [34, 35]. Precise memory safety enforcement [34, 36, 25] tracks the based-on information for each pointer in a program, to check the safety of each pointer dereference according to the definition above; the detection of an unsafe dereference aborts the program.

3.1 The Code-Pointer Integrity (CPI) Property

A program execution satisfies the code-pointer integrity property iff all its dereferences that either dereference or access sensitive pointers are safe. *Sensitive pointers* are code pointers and pointers that may later be used to access sensitive pointers. Note that the sensitive pointer definition is recursive, as illustrated in Fig. 1. According to case (iv) of the based-on definition above, dereferencing a pointer to a pointer will correspondingly propagate the based-on information; e.g., an expression *p = &q copies the result of &q, which is a pointer based on q, to a location pointed to by p, and associates the based-on metadata with that location. Hence, the integrity of the based-on metadata associated with sensitive pointers requires that pointers used to update sensitive pointers be sensitive as well (we discuss implications of relaxing this definition in §3.3). The notion of a sensitive pointer is dynamic. For example, a void* pointer 2 in Fig. 1 is sensitive when it points at another sensitive pointer at run time, but it is not sensitive when it points to an integer.

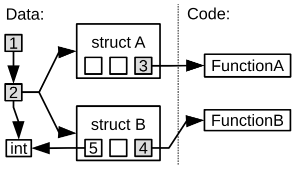


Figure 1: CPI protects code pointers 3 and 4 and pointers 1 and 2 (which may access pointers 3 and 4 indirectly). Pointer 2 of type void* may point to different objects at different times. The int* pointer 5 and non-pointer data locations are not protected.

A memory-safe program execution trivially satisfies the CPI property, but memory-safety instrumentation typically has high runtime overhead, e.g., $\geq 2\times$ in state-of-the-art implementations [35]. Our observation is that only a small subset of all pointers are responsible for making control-flow transfers, and so, by enforcing memory safety only for control-sensitive data (and thus incurring no overhead for all other data), we obtain important security guarantees while keeping the cost of enforcement low. This is analogous to the control-plane/data-plane separation in network routers and modern servers [5], with CPI ensuring the safety of data that influences, directly or indirectly, the control plane.

Determining precisely the set of pointers that are sensitive can only be done at run time. However, the CPI property can still be enforced using any over-approximation of this set, and such over-approximations can be obtained at compile time, using static analysis.

3.2 The CPI Enforcement Mechanism

We now describe a way to retrofit the CPI property into a program P using a combination of static instrumentation and runtime support. Our approach consists of a *static analysis* pass that identifies all sensitive pointers in P and all instructions that operate on them (§3.2.1), an *instrumentation* pass that rewrites P to "protect" all sensitive pointers, i.e., store them in a separate, safe memory region and associate, propagate, and check their based-on metadata (§3.2.2), and an instruction-level *isolation* mechanism that prevents non-protected memory operations from accessing the safe region (§3.2.3). For performance reasons, we handle return addresses stored on the stack separately from the rest of the code pointers using a *safe stack* mechanism (§3.2.4).

3.2.1 CPI Static Analysis

We determine the set of sensitive pointers using type-based static analysis: a pointer is sensitive if its type is sensitive. Sensitive types are: pointers to functions, pointers to sensitive types, pointers to composite types (such as struct or array) that contains one or more members of sensitive types, or universal pointers (i.e., void*, char* and opaque pointers to forward-declared structs or classes). A programmer could additionally indicate, if desired, other types to be considered sensitive, such as struct ucred used in the FreeBSD kernel to store process UIDs and jail information. All code pointers that a compiler or runtime creates implicitly (such as return addresses, C++ virtual table pointers, and setjmp buffers) are sensitive as well.

Once the set of sensitive pointers is determined, we use static analysis to find all program instructions that manipulate these pointers. These instructions include pointer dereferences, pointer arithmetic, and memory (de-)allocation operations that calls to either (i) corresponding standard library functions, (ii) C++ new/delete operators, or (iii) manually annotated custom allocators.

The derived set of sensitive pointers is over-approximate: it may include universal pointers that never end up pointing to sensitive values at runtime. For instance, the C/C++ standard allows char* pointers to point to objects of any type, but such pointers are also used for C strings. As a heuristic, we assume that char* pointers that are passed to the standard libc string manipulation functions or that are assigned to point to string constants are not universal. Neither the over-approximation nor the char* heuristic affect the security guarantees provided by CPI: over-approximation merely introduces extra overhead, while heuristic errors may result in false violation reports (though we never observed any in practice).

Memory manipulation functions from libc, such as memset or memcpy, could introduce a lot of overhead in CPI: they take void* arguments, so a libc compiled with CPI would instrument all accesses inside the functions, regardless of whether they are operating on sensitive data or not. CPI's static analysis instead detects such cases by analyzing the real types of the arguments prior to being cast to void*, and the subsequent instrumentation pass handles them separately using type-specific versions of the corresponding memory manipulation functions.

We augmented type-based static analysis with a data-flow analysis that handles most practical cases of unsafe pointer casts and casts between pointers and integers. If a value v is ever cast to a sensitive pointer type within the function being analyzed, or is passed as an argument or returned to another function where it is cast to a sensitive pointer, the analysis considers v to be sensitive as well. This analysis may fail when the data flow between v and its cast to a sensitive pointer type cannot be fully recovered statically, which might cause false violation reports (we have not observed any during our evaluation). Such casts are a common problem for all pointer-based memory safety mechanisms for C/C++ that do not require source code modifications [34].

A key benefit of CPI is its selectivity: the number of pointer operations deemed to be sensitive is a small fraction of all pointer operations in a program. As we show in §5, for SPEC CPU2006, the CPI type-based analysis identifies for instrumentation 6.5% of all pointer accesses; this translates into a reduction of performance overhead of $16-44\times$ relative to full memory safety.

Nevertheless, we still think CPI can benefit from more sophisticated analyses. CPI can leverage any kind of *points-to* static analysis, as long as it provides an over-approximate set of sensitive pointers. For instance, when extending CPI to also protect select non-code-pointer data, we think DSA [27, 28] could prove more effective.

3.2.2 CPI Instrumentation

CPI instruments a program in order to (i) ensure that all sensitive pointers are stored in a safe region, (ii) create and propagate metadata for such pointers at runtime, and (iii) check the metadata on dereferences of such pointers.

In terms of memory layout, CPI introduces a safe region in addition to the regular memory region (Fig. 2). Storage space for sensitive pointers is allocated in both the safe region (the *safe pointer store*) and the regular region (as usual); one of the two copies always remains unused. This is necessary for universal pointers (e.g., void*), which could be stored in either region depending on whether they are sensitive at run time or not, and also helps to avoid some compatibility issues that arise from the change in memory layout. The address in regular memory is used as an offset to to look up the value of a sensitive pointer in the safe pointer store.

The *safe pointer store* maps the address &p of sensitive pointer p, as allocated in the regular region, to the value of p and associated metadata. The metadata for p describes the target object on which p is based: lower and upper address bounds of the object, and a temporal id (see Fig. 2). The layout of the safe pointer store is similar to metadata storage in SoftBounds+CETS [35], except that CPI *also* stores the value of p in the safe pointer store. Combined with the isolation of the safe region (§3.2.3), this allows CPI to guarantee full memory safety of all sensitive pointers without having to instru-

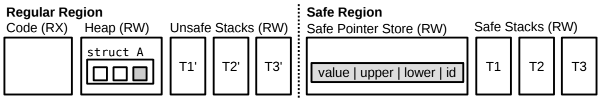


Figure 2: CPI memory layout: The safe region contains the safe pointer store and the safe stacks. The location of a sensitive pointer on the left (shaded) remains unused, while the value of this pointer and its metadata are stored in the safe pointer store. The safe stacks T_1, T_2, T_3 have corresponding stacks T_1', T_2', T_3' in regular memory to allocate unsafe stack objects.

ment all pointer operations.

The instrumentation step changes instructions that operate on sensitive pointers, as found by CPI's static analysis, to create and propagate the metadata directly following the based-on definition in §3.1. Instructions that explicitly take addresses of a statically allocated memory object or a function, allocate a new object on the heap, or take an address of a sub-object are instrumented to create metadata that describe the corresponding object. Instructions that compute pointer expressions are instrumented to propagate the metadata accordingly. Instructions that load or store sensitive pointers to memory are replaced with CPI intrinsic instructions (§3.2.3) that load or store both the pointer values and their metadata from/to the safe pointer store. In principle, call and return instructions also store and load code pointers, and so would need to be instrumented, but we instead protect return addresses using a safe stack (§3.2.4).

Every dereference of a sensitive pointer is instrumented to check at runtime whether it is safe, using the metadata associated with the pointer being dereferenced. Together with the restricted access to the safe region, this results in precise memory safety for all sensitive pointers.

Universal pointers (void* and char*) are stored in either the safe pointer store or the regular region, depending on whether they are sensitive at runtime or not. CPI instruments instructions that cast from non-sensitive to universal pointer types to assign special "invalid" metadata (e.g., with lower bound greater than the upper bound) for the resulting universal pointers. These pointers, as a result, would never be allowed to access the safe region. CPI intrinsics for universal pointers would only store a pointer in the safe pointer store if it had valid metadata, and only load it from the safe pointer store if it contained valid metadata for that pointer; otherwise, they would store/load from the regular region.

CPI can be configured to simultaneously store protected pointers in both the safe pointer store and regular regions, and check whether they match when loading them. In this debug mode, CPI detects all *attempts* to hijack control flow using non-protected pointer errors; in the default mode, such attempts are silently prevented. This debug mode also provides better compatibility with non-instrumented code that may read protected pointers (for example, callback addresses) but not write them.

Modern compilers contain powerful static analysis passes that can often prove statically that certain memory accesses are always safe. The CPI instrumentation pass precedes compiler optimizations, thus allowing them to potentially optimize away some of the inserted checks while preserving the security guarantees.

3.2.3 Isolating the Safe Region

The safe region can only be accessed via CPI intrinsic instructions, and they properly handle pointer metadata and the safe stack (§3.2.4). The mechanism for achieving this isolation is architecture-dependent.

On x86-32, we rely on hardware segment protection. We make the safe region accessible through a dedicated segment register, which is otherwise unused, and configure limits for all other segment registers to make the region inaccessible through them. The CPI intrinsics are then turned into code that uses the dedicated register and ensures that no other instructions in the program use that register. The segment registers are configured by the program loader, whose integrity we assume in our threat model; we also prevent the program from reconfiguring the segment registers via system calls. None of the programs we evaluated use the segment registers.

On x86-64, CPI relies on the fact that no addresses pointing into the safe region are ever stored in the regular region. This architecture no longer enforces the segment limits, however it still provides two segment registers with configurable base addresses. Similarly to x86-32, we use one of these registers to point to the safe region, however, we choose the base address of the safe region at random and rely on preventing access to it through information hiding. Unlike classic ASLR though, our hiding is leak-proof: since the objects in the safe region are indexed by addresses allocated for them in the regular region, no addresses pointing into the safe region are ever stored in regular memory at any time during execution. The 48-bit address space of modern x86-64 CPUs makes guessing the safe region address impractical, because most failed guessing attempts would crash the program, and such frequent crashes can easily be detected by other means.

Other architectures could use randomization-based protection as well, or rely on precise software fault isolation (SFI) [11]. SFI requires that all memory operations in a program are instrumented, but the instrumentation is lightweight: it could be as small as a single and operation if the safe region occupies the entire upper half of the address space of a process. In our experiments, the additional overhead introduced by SFI was less than 5%.

Since sensitive pointers form a small fraction of all data stored in memory, the safe pointer store is highly sparse. To save memory, it can be organized as a hash table, a multi-level lookup table, or as a simple array relying on the sparse address space support of the underlying OS. We implemented and evaluated all three versions, and we discuss the fastest choice in §4.

In the future, we plan to leverage Intel MPX [24] for implementing the safe region, as described in §4.

3.2.4 The Safe Stack

CPI treats the stack specially, in order to reduce performance overhead and complexity. This is primarily because the stack hosts values that are accessed frequently, such as return addresses that are code pointers accessed on every function call, as well as spilled registers (temporary values that do not fit in registers and compilers store on the stack). Furthermore, tracking which of these values will end up at run time in memory (and thus need to be protected) vs. in registers is difficult, as the compiler decides which registers to spill only during late stages of code generation, long after CPI's instrumentation pass.

A key observation is that the safety of most accesses to stack objects can be checked statically during compilation, hence such accesses require no runtime checks or metadata. Most stack frames contain only memory objects that are accessed exclusively within the corresponding function and only through the stack pointer register with a constant offset. We therefore place all such proven-safe objects onto a *safe stack* located in the safe region. The safe stack can be accessed without any checks. For functions that have memory objects on their stack that do require checks (e.g., arrays or objects whose address is passed to other functions), we allocate separate stack frames in the regular memory region. In our experience, less than 25% of functions need such additional stack frames (see Table 2). Furthermore, this fraction is much smaller among short functions, for which the overhead of setting up the extra stack frame is non-negligible.

The safe stack mechanism consists of a static analysis pass, an instrumentation pass, and runtime support. The analysis pass identifies, for every function, which objects in its stack frame are guaranteed to be accessed safely and can thus be placed on the safe stack; return addresses and spilled registers always satisfy this criterion. For the objects that do not satisfy this criterion, the instrumentation pass inserts code that allocates a stack frame for these objects on the regular stack. The runtime support allocates regular stacks for each thread and can be implemented either as part of the threading library, as we did on FreeBSD, or by intercepting thread create/destroy, as we did on Linux. CPI stores the regular stack pointer inside the thread control block, which is pointed to by one of the segment registers and can thus be accessed with a single memory read or write.

Our safe stack layout is similar to double stack approaches in ASR [6] and XFI [18], which maintain a separate stack for arrays and variables whose addresses are taken. However, we use the safe stack to enforce the CPI property instead of implementing software fault isolation. The safe stack is also comparable to language-based approaches like Cyclone [25] or CCured [36] that simply allocate these objects on the heap, but our approach has significantly lower performance overhead.

Compared to a shadow stack like in CFI [1], which duplicates return instruction pointers outside of the attacker's access, the CPI safe stack presents several advantages: (i) all return instruction pointers and most local variables are protected, whereas a shadow stack only protects return instruction pointers; (ii) the safe stack is compatible with uninstrumented code that uses just the regular stack, and it directly supports exceptions, tail calls, and signal handlers; (iii) the safe stack has near-zero performance overhead (§5.2), because only a handful of functions require extra stack frames, while a shadow stack allocates a shadow frame for every function call.

The safe stack can be employed independently from CPI, and we believe it can replace stack cookies [14] in modern compilers. By providing precise protection of all return addresses (which are the target of ROP attacks today), spilled registers, and some local variables, the safe stack provides substantially stronger security than stack cookies, while incurring equal or lower performance overhead and deployment complexity.

3.3 Code-Pointer Separation (CPS)

The code-pointer separation property trades some of CPI's security guarantees for reduced runtime overhead. This is particularly relevant to C++ programs with many virtual functions, where the fraction of sensitive pointers instrumented by CPI can become high, since every pointer to an object that contains virtual functions is sensitive. We found that, on average, CPS reduces overhead by 4.3× (from 8.4% for CPI down to 1.9% for CPS), and in some cases by as much as an order of magnitude.

CPS further restricts the set of protected pointers to code pointers only, leaving pointers that point to code pointers uninstrumented. We additionally restrict the definition of based-on by requiring that a code pointer be based only on a control flow destination. This restriction prevents attackers from "forging" a code pointer from a value of another type, but still allows them to trick the program into reading or updating wrong code pointers.

CPS is enforced similarly to CPI, except (i) for the criteria used to identify sensitive pointers during static analysis, and (ii) that CPS does not need any metadata. Control-flow destinations (pointed to by code pointers) do not have bounds, because the pointer value must always match the destination exactly, hence no need for bounds metadata. Furthermore, they are typically static, hence do not need temporal metadata either (there are a few rare exceptions, like unloading a shared library, which are handled separately). This reduces the size of the safe region and the number of memory accesses when loading or storing code pointers. If the safe region is organized as a simple array, a CPS-instrumented program performs essentially the same number of memory accesses when loading or storing code pointers as a non-instrumented one; the only difference is that the pointers are being loaded or stored from the safe pointer store instead of their original location (universal pointer load or store instructions still introduce one extra memory access per such instruction). As a result, CPS can be enforced with low performance overhead.

CPS guarantees that (i) code pointers can only be stored to or modified in memory by code pointer store instructions, and (ii) code pointers can only be loaded by code pointer load instructions from memory locations to which previously a code pointer store instruction stored a value. Combined with the safe stack, CPS precisely protects return addresses. CPS is stronger than most CFI implementations [1, 54, 53], which allow any vulnerable instruction in a program to modify any code pointer; they only check that the value of a code pointer (when used in an indirect control transfer) points to a function defined in a program (for function pointers) or directly follows a call instruction (for return addresses). CPS guarantee (i) above restricts the attack surface, while guarantee (ii) restricts the attacker's flexibility by limiting the set of locations to which the control can be redirected—the set includes only entry points of functions whose addresses were explicitly taken by the program.

To illustrate this difference, consider the case of the Perl interpreter, which implements its opcode dispatch by representing internally a Perl program as a sequence of function pointers to opcode handlers and then calling in its main execution loop these function pointers one by one. CFI statically approximates the set of legitimate control-flow targets, which in this case would include all possible Perl opcodes. CPS however permits only calls through function pointers that are actually assigned. This means that a memory bug in a CFI-protected Perl interpreter may permit an attacker to divert control flow and execute any Perl opcode, whereas in a CPS-protected Perl interpreter the attacker could at most execute an opcode that exists in the running Perl program.

CPS provides strong control-flow integrity guarantees and incurs low overhead (§5). We found that it prevents all recent attacks designed to bypass CFI [19, 15, 9]. We consider CPS to be a solid alternative to CPI in those cases when CPI's (already low) overhead seems too high.

4 Implementation

We implemented a CPI/CPS enforcement tool for C/C++, called Levee, on top of the LLVM 3.3 compiler infrastructure [30], with modifications to LLVM libraries, the clang compiler, and the compiler-rt runtime. To use Levee, one just needs to pass additional flags to the compiler to enable CPI (-fcpi), CPS (-fcps), or safe-stack protection (-fstack-protector-safe). Levee works on unmodified programs and supports Linux, FreeBSD, and Mac OS X in both 32-bit and 64-bit modes.

Levee can be downloaded from the project homepage `http://levee.epfl.ch`, and we plan to push our changes to the upstream LLVM.

Analysis and instrumentation passes: We implemented the static analysis and instrumentation for CPI as two LLVM passes, directly following the design from §3.2.1 and §3.2.2. The LLVM passes operate on the LLVM intermediate representation (IR), which is a low-level strongly-typed language-independent program representation tailored for static analyses and optimization purposes. The LLVM IR is generated from the C/C++ source code by clang, which preserves most of the type information that is required by our analysis, with a few corner cases. For example, in certain cases, clang does not preserve the original types of pointers that are cast to void* when passing them as an argument to memset or similar functions, which is required for the memset-related optimizations discussed in §3.2.2. The IR also does not distinguish between void* and char* (represents both as i8*), but this information is required for our string pointers detection heuristic. We augmented clang to always preserve such type information as LLVM metadata.

Safe stack instrumentation pass: The safe stack instrumentation targets functions that contain on-stack memory objects that cannot be put on the safe stack. For such functions, it allocates a stack frame on the unsafe stack and relocates corresponding variables to that frame.

Given that most of the functions do not need an unsafe stack, Levee uses the usual stack pointer (rsp register on x86-64) as the safe stack pointer, and stores the unsafe stack pointer in the thread control block, which is accessible directly through one of the segment registers. When needed, the unsafe stack pointer is loaded into an IR local value, and Levee relies on the LLVM register allocator to pick the register for the unsafe stack pointer. Levee explicitly encodes unsafe stack operations as IR instructions that manipulate an unsafe stack pointer; it leaves all operations that use a safe stack intact, letting the LLVM code generator manage them. Levee performs these changes as a last step before code generation (directly replacing LLVM's stack-cookie protection pass), thus ensuring that it operates on the final stack layout.

Certain low-level functions modify the stack pointer directly. These functions include setjmp/longjmp and exception handling functions (which store/load the stack pointer), and thread create/destroy functions, which allocate/free stacks for threads. On FreeBSD we provide full-system CPI, so we directly modified these functions to support the dual stacks. On Linux, our instrumentation pass finds setjmp/longjmp and exception handling functions in the program and inserts required instrumentation at their call sites, while thread create/destroy functions are intercepted and handled by the Levee runtime.

Runtime support library: Most of the instrumentation by the above passes are added as intrinsic function calls, such as cpi_ptr_store() or cpi_memcpy(), which are implemented by Levee's runtime support library (a part of compiler-rt). This design cleanly separates the safe pointer store implementation from the instrumentation pass. In order to avoid the overhead associated with extra function calls, we ensure that some of the runtime support functions are always inlined. We compile these functions into LLVM bitcode and instruct clang to link this bitcode into every object file it compiles. Functions that are called rarely (e.g., cpi_abort(), called when a CPI violation is detected) are never inlined, in order to reduce the instruction cache footprint of the instrumentation.

We implemented and benchmarked several versions of the safe pointer store map in our runtime support library: a simple array, a two-level lookup table, and a hashtable. The array implementation relies on the sparse address space support of the underlying OS. Initially we found it to perform poorly on Linux, due to many page faults (especially at startup) and additional TLB pressure. Switching to superpages (2 MB on Linux) made this simple table the fastest implementation of the three.

Binary level functionality: Some code pointers in binaries are generated by the compiler and/or linker, and cannot be protected on the IR level. Such pointers include the ones in jump tables, exception handler tables, and the global offset table. Bounds checks for the jump tables and the exception handler tables are already generated by LLVM anyway, and the tables themselves are placed in read-only memory, hence cannot be overwritten. We rely on the standard loader's support for read-only global offset tables, using the existing RTLD_NOW flag.

Limitations: The CPI design described in §3 includes both spatial and temporal memory safety enforcement for sensitive pointers, however our current prototype implements spatial memory safety only. It can be easily extended to enforce temporal safety by directly applying the technique described in [35] for sensitive pointers.

Levee currently supports Linux, FreeBSD and Mac OS user-space applications. We believe Levee can be ported to protect OS kernels as well. Related technical challenges include integration with the kernel memory management subsystem and handling of inline assembly.

CPI and CPS require instrumenting all code that manipulates sensitive pointers; non-instrumented code can cause unnecessary aborts. Non-instrumented code could come from external libraries compiled without Levee, inline assembly, or dynamically generated code. Levee can be configured to simultaneously store sensitive pointers in both the safe and the regular regions, in which case non-instrumented code works fine as long as it only reads sensitive pointers but doesn't write them.

Inline assembly and dynamically generated code can still update sensitive pointers if instrumented with appropriate calls to the Levee runtime, either manually by a programmer or directly by the code generator.

Dynamically generated code (e.g., for JIT compilation) poses an additional problem: running the generated code requires making writable pages executable, which violates our threat model (this is a common problem for most control-flow integrity mechanisms). One solution is to use hardware or software isolation mechanisms to isolate the code generator from the code it generates.

Sensitive data protection: Even though the main focus of CPI is control-flow hijack protection, the same technique can be applied to protect other types of sensitive data. Levee can treat programmer-annotated data types as sensitive and protect them just like code pointers. CPI could also selectively protect individual program variables (as opposed to types), however it would require replacing the type-based static analysis described in §3.2.1 with data-based points-to analysis such as DSA [27, 28].

Future MPX-based implementation: Intel announced a hardware extension, Intel MPX, to be used for hardware-enforced memory safety [23]. It is proposed as a testing tool, probably due to the associated overhead; no overhead numbers are available at the time of writing.

We believe MPX (or similar) hardware can be repurposed to enforce CPI with lower performance overhead than our existing software-only implementation. MPX provides special registers to store bounds along with instructions to check them, and a hardware-based implementation of a pointer metadata store (analogous to the safe pointer store in our design), organized as a two-level lookup table. Our implementation can be adapted to use these facilities once MPX-enabled hardware becomes available. We believe that a hardware-based CPI implementation can reduce the overhead of a software-only CPI in much the same way as HardBound [16] or Watchdog [33] reduced the overhead of SoftBound.

Adopting MPX for CPI might require implementing metadata loading logic in software. Like CPI, MPX also stores the pointer value together with the metadata. However, being a testing tool, MPX chooses compatibility with non-instrumented code over security guarantees: it uses the stored pointer value to check whether the original pointer was modified by non-instrumented code and, if yes, resets the bounds to $[0,\infty]$. In contrast, CPI's guarantees depend on preventing any non-instrumented code from ever modifying sensitive pointer values.

5 Evaluation

In this section we evaluate Levee's effectiveness, efficiency, and practicality. We experimentally show that both CPI and CPS are 100% effective on RIPE, the most recent attack benchmark we are aware of (§5.1). We evaluate the efficiency of CPI, CPS, and the safe stack on SPEC CPU2006, and find average overheads of 8.4%, 1.9%, and 0% respectively (§5.2). To demonstrate practicality, we recompile with CPI/CPS/ safe stack the base FreeBSD plus over 100 packages and report results on several benchmarks (§5.3).

We ran our experiments on an Intel Xeon E5-2697 with 24 cores @ 2.7GHz in 64-bit mode with 512GB RAM. The SPEC benchmarks ran on an Ubuntu Precise Pangolin (12.04 LTS), and the FreeBSD benchmarks in a KVM-based VM on this same system.

5.1 Effectiveness on the RIPE Benchmark

We described in §3 the security guarantees provided by CPI, CPS, and the safe stack based on their design; to experimentally evaluate their effectiveness, we use the RIPE [49] benchmark. This is a program with many different security vulnerabilities and a set of 850 exploits that attempt to perform control-flow hijack attacks on the program using various techniques.

Levee deterministically prevents all attacks, both in CPS and CPI mode; when using only the safe stack, it prevents all stack-based attacks. On vanilla Ubuntu 6.06, which has no built-in defense mechanisms, 833–848 exploits succeed when Levee is not used (some succeed probabilistically, hence the range). On newer systems, fewer exploits succeed, due to built-in protection mechanisms, changes in the run-time layout, and compatibility issues with the RIPE benchmark. On vanilla Ubuntu 13.10, with all protections (DEP, ASLR, stack cookies) disabled, 197–205 exploits succeed. With all protections enabled, 43–49 succeed. With CPS or CPI, none do.

The RIPE benchmark only evaluates the effectiveness of preventing existing attacks; as we argued in §3 and according to the proof outlined in Appendix A, CPI renders all (known and unknown) memory corruption-based control-flow hijack attacks impossible.

5.2 Efficiency on SPEC CPU2006 Benchmarks

In this section we evaluate the runtime overhead of CPI, CPS, and the safe stack. We report numbers on all SPEC CPU2006 benchmarks written in C and C++ (our prototype does not handle Fortran). The results are summarized in Table 1 and presented in detail in Fig. 3. We also compare Levee to two related approaches, SoftBound [34] and control-flow integrity [1, 54, 53].

CPI performs well for most C benchmarks, however it can incur higher overhead for programs written in C++. This overhead is caused by abundant use of pointers to C++ objects that contain virtual function tables—such pointers are sensitive for CPI, and so all operations on them are instrumented. Same reason holds for gcc: it embeds function pointers in some of its data structures and then uses pointers to these structures frequently.

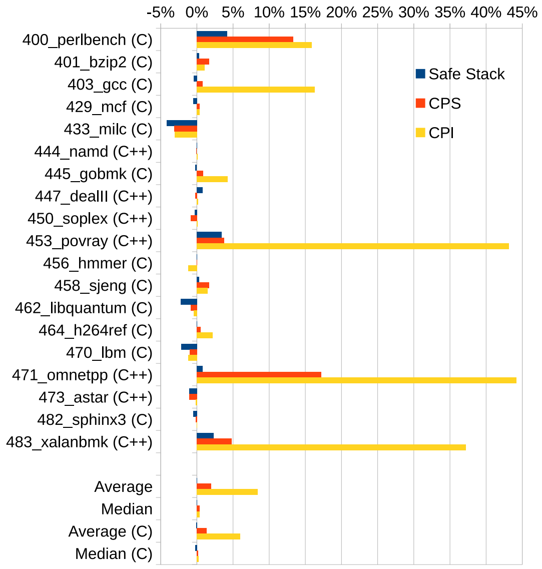


Figure 3: Levee performance for SPEC CPU2006, under three configurations: full CPI, CPS only, and safe stack only.

	Safe Stack	CPS	CPI
Average (C/C++)	0.0%	1.9%	8.4%
Median (C/C++)	0.0%	0.4%	0.4%
Maximum (C/C++)	4.1%	17.2%	44.2%
Average (C only)	-0.4%	1.2%	2.9%
Median (C only)	-0.3%	0.5%	0.7%
Maximum (C only)	4.1%	13.3%	16.3%

Table 1: Summary of SPEC CPU2006 performance overheads.

The next-most important source of overhead are libc memory manipulation functions, like memset and memcpy. When our static analysis cannot prove that a call to such a function uses as arguments only pointers to non-sensitive data, Levee replaces the call with one to a custom version of an equivalent function that checks the safe pointer store for each updated/copied word, which introduces overhead. We expect to remove some of this overhead using improved static analysis and heuristics.

CPS averages 1.2–1.8% overhead, and exceeds 5% on only two benchmarks, omnetpp and perlbench. The former is due to the large number of virtual function calls occurring at run time, while the latter is caused by a specific way in which perl implements its opcode dispatch: it internally represents a program as a sequence of function pointers to opcode handlers, and its main execution loop calls these function pointers one after the other. Most other interpreters use a switch for opcode dispatch.

Safe stack provided a surprise: in 9 cases (out of

19), it improves performance instead of hurting it; in one case (namd), the improvement is as high as 4.2%, more than the overhead incurred by CPI and CPS. This is because objects that end up being moved to the regular (unsafe) stack are usually large arrays or variables that are used through multiple stack frames. Moving such objects away from the safe stack increases the locality of frequently accessed values on the stack, such as CPU register values temporarily stored on the stack, return addresses, and small local variables.

The safe stack overhead exceeds 1% in only three cases, perlbench, xalanbmk, and povray. We studied the disassembly of the most frequently executed functions that use unsafe stack frames in these programs and found that some of the overhead is caused by inefficient handling of the unsafe stack pointer by LLVM's register allocator. Instead of keeping this pointer in a single register and using it as a base for all unsafe stack accesses, the program keeps moving the unsafe stack pointer between different registers and often spills it to the (safe) stack. We believe this can be resolved by making the register allocator algorithm aware of the unsafe stack pointer.

In contrast to the safe stack, stack cookies deployed today have an overhead of up to 5%, and offer strictly weaker protection than our safe stack implementation.

The data structures used for the safe stack and the safe memory region result in memory overhead compared to a program without protection. We measure the memory overhead when using either a simple array or a hash table. For SPEC CPU2006 the median memory overhead for the safe stack is 0.1%; for CPS the overhead is 2.1% for the hash table and 5.6% for the array; and for CPI the overhead is 13.9% for the hash table and 105% for the array. We did not optimize the memory overhead yet and believe it can be improved in future prototypes.

In Table 2 we show compilation statistics for Levee. The first column shows that only a small fraction of all functions require an unsafe stack frame, confirming our hypothesis from §3.2.4. The other two columns confirm the key premises behind our approach, namely that CPI requires much less instrumentation than full memory safety, and CPS needs much less instrumentation than CPI. The numbers also correlate with Fig. 3.

Comparison to SoftBound: We compare with SoftBound [34] on the SPEC benchmarks. We cannot fairly reuse the numbers from [34], because they are based on an older version of SPEC. In Table 3 we report numbers for the four C/C++ SPEC benchmarks that can compile with the current version of SoftBound. This comparison confirms our hypothesis that CPI requires significantly lower overhead compared to full memory safety.

Theoretically, CPI suffers from the same compatibility issues (e.g., handling unsafe pointer casts) as pointer-based memory safety. In practice, such issues arise much less frequently for CPI, because CPI instruments much fewer pointers. Many of the SPEC benchmarks either don't compile or terminate with an error when instrumeted by SoftBound, which illustrates the practical impact of this difference.

Benchmark	**FN$_{UStack}$**	**MO$_{CPS}$**	**MO$_{CPI}$**
400_perlbench	15.0%	1.0%	13.8%
401_bzip2	27.2%	1.3%	1.9%
403_gcc	19.9%	0.3%	6.0%
429_mcf	50.0%	0.5%	0.7%
433_milc	50.9%	0.1%	0.7%
444_namd	75.8%	0.6%	1.1%
445_gobmk	10.3%	0.1%	0.4%
447_dealII	12.3%	6.6%	13.3%
450_soplex	9.5%	4.0%	2.5%
453_povray	26.8%	0.8%	4.7%
456_hmmer	13.6%	0.2%	2.0%
458_sjeng	50.0%	0.1%	0.1%
462_libquantum	28.5%	0.4%	2.3%
464_h264ref	20.5%	1.5%	2.8%
470_lbm	16.6%	0.6%	1.5%
471_omnetpp	6.9%	10.5%	36.6%
473_astar	9.0%	0.1%	3.2%
482_sphinx3	19.7%	0.1%	4.6%
483_xalancbmk	17.5%	17.5%	27.1%

Table 2: Compilation statistics for Levee: FN$_{UStack}$ lists what fraction of functions need an unsafe stack frame; MO$_{CPS}$ and MO$_{CPI}$ show the fraction of memory operations instrumented for CPS and CPI, respectively.

Benchmark	**Safe Stack**	**CPS**	**CPI**	**SoftBound**
401_bzip2	0.3%	1.2%	2.8%	90.2%
447_dealII	0.8%	-0.2%	3.7%	60.2%
458_sjeng	0.3%	1.8%	2.6%	79.0%
464_h264ref	0.9%	5.5%	5.8%	249.4%

Table 3: Overhead of Levee and SoftBound on SPEC programs that compile and run errors-free with SoftBound.

Comparison to control-flow integrity (CFI): The average overhead for compiler-enforced CFI is 21% for a subset of the SPEC CPU2000 benchmarks [1] and 5-6% for MCFI [39] (without stack pointer integrity). CCFIR [53] reports an overhead of 3.6%, and binCFI [54] reports 8.54% for SPEC CPU2006 to enforce a weak CFI property with globally merged target sets. WIT [3], a source-based mechanism that enforces both CFI and write integrity protection, has 10% overhead[1].

At less than 2%, CPS has the lowest overhead among all existing CFI solutions, while providing stronger protection guarantees. Also, CPI's overhead is bested only by CCFIR. However, unlike any CFI mechanism, CPI guarantees the impossibility of any control-flow hijack attack based on memory corruptions. In contrast, there

[1] We were unable to find open-source implementations of compiler-based CFI, so we can only compare to published overhead numbers.

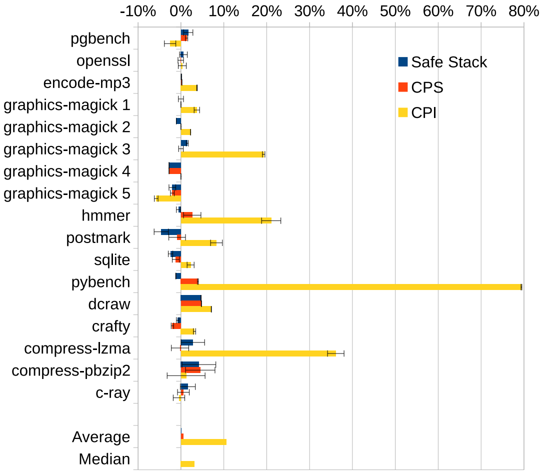


Figure 4: Performance overheads on FreeBSD (Phoronix).

exist successful attacks against CFI [19, 15, 9]. While neither of these attacks are possible against CPI by construction, we found that, in practice, neither of them would work against CPS either. We further discuss conceptual differences between CFI and CPI in §6.

5.3 Case Study: A Safe FreeBSD Distribution

Having shown that Levee is both effective and efficient, we now evaluate the feasibility of using Levee to protect an entire operating system distribution, namely FreeBSD 10. We rebuilt the base system—base libraries, development tools, and services like bind and openssh—plus more than 100 packages (including apache, postgresql, php, python) in four configurations: CPI, CPS, Safe Stack, and vanilla. FreeBSD 10 uses LLVM/clang as its default compiler, while some core components of Linux (e.g., glibc) cannot be built with clang yet. We integrated the CPI runtime directly into the C library and the threading library. We have not yet ported the runtime to kernel space, so the OS kernel remained uninstrumented.

We evaluated the performance of the system using the Phoronix test suite [41], a widely used comprehensive benchmarking platform for operating systems. We chose the "server" setting and excluded benchmarks marked as unsupported or that do not compile or run on recent FreeBSD versions. All benchmarks that compiled and worked on vanilla FreeBSD also compiled and worked in the CPI, CPS and Safe Stack versions.

Fig. 4 shows the overhead of CPI, CPS and the safe-stack versions compared to the vanilla version. The results are consistent with the SPEC results presented in §5.2. The Phoronix benchmarks exercise large parts of the system and some of them are multi-threaded, which introduces significant variance in the results, especially when run on modern hardware. As Fig. 4 shows, for many benchmarks the overheads of CPS and the safe stack are within the measurement error.

Benchmark	Safe Stack	CPS	CPI
Static page	1.7%	8.9%	16.9%
Wsgi test page	1.0%	4.0%	15.3%
Dynamic page	1.4%	15.9%	138.8%

Table 4: Throughput benchmark for web server stack (FreeBSD + Apache + SQLite + mod_wsgi + Python +Django).

We also evaluated a realistic usage model of the FreeBSD system as a web server. We installed Mezzanine, a content management system based on Django, which uses Python, SQLite, Apache, and mod_wsgi. We used the Apache ab tool to benchmark the throughput of the web server. The results are summarized in Table 4.

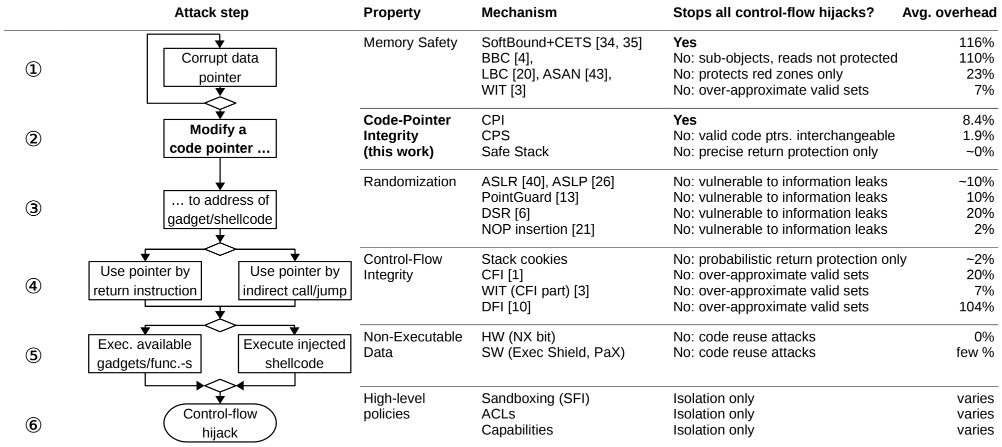
The CPI overhead for a dynamic page generated by Python code is much larger then we expected, but consistent with suspiciously high overhead of the pybench benchmark in Fig. 4. We think it might be caused by the use of some C constructs in the Python interpreter that are not yet handled well by our optimization heuristics, e.g., emulating C++ inheritance in C. We believe the performance might be improved in this case by extending the heuristics to recognize such C constructs.

6 Related Work

A variety of defense mechanisms have been proposed to-date to answer the increasing challenge of control-flow hijack attacks. Fig. 5 compares the design of the different protection approaches to our approach.

Enforcing memory safety ensures that no dangling or out-of-bounds pointers can be read or written by the application, thus preventing the attack in its first step. Cyclone [25] and CCured [36] extend C with a safe type system to enforce memory safety features. These approaches face the problem that there is a large (unported) legacy code base. In contrast, CPI and CPS both work for unmodified C/C++ code. SoftBound [34] with its CETS [35] extension enforces *complete* memory safety at the cost of 2× – 4× slowdown. Tools with less overhead, like BBC [4], only *approximate* memory safety. LBC [20] and Address Sanitizer [43] detect continuous buffer overflows and (probabilistically) indexing errors, but can be bypassed by an attacker who avoids the red zones placed around objects. Write integrity testing (WIT) [3] provides spatial memory safety by restricting pointer writes according to points-to sets obtained by an over-approximate static analysis (and is therefore limited by the static analysis). Other techniques [17, 2] enforce type-safe memory reuse to mitigate attacks that exploit temporal errors (use after frees).

CPI by design enforces spatial and temporal memory safety for a subset of data (code pointers) in Step 2 of Fig. 5. Our Levee prototype currently enforces spatial memory safety and may be extended to enforce temporal memory safety as well (e.g., how CETS extends Soft-


Attack step
①
Corrupt data pointer
②
Modify a code pointer …
③
… to address of gadget/shellcode
④
Use pointer by return instruction
Use pointer by indirect call/jump
⑤
Exec. available gadgets/func.-s
Execute injected shellcode
⑥
Control-flow hijack

Property	Mechanism	Stops all control-flow hijacks?	Avg. overhead
Memory Safety	SoftBound+CETS [34, 35]	**Yes**	116%
	BBC [4],	No: sub-objects, reads not protected	110%
	LBC [20], ASAN [43],	No: protects red zones only	23%
	WIT [3]	No: over-approximate valid sets	7%
Code-Pointer Integrity (this work)	CPI	**Yes**	8.4%
	CPS	No: valid code ptrs. interchangeable	1.9%
	Safe Stack	No: precise return protection only	~0%
Randomization	ASLR [40], ASLP [26]	No: vulnerable to information leaks	~10%
	PointGuard [13]	No: vulnerable to information leaks	10%
	DSR [6]	No: vulnerable to information leaks	20%
	NOP insertion [21]	No: vulnerable to information leaks	2%
Control-Flow Integrity	Stack cookies	No: probabilistic return protection only	~2%
	CFI [1]	No: over-approximate valid sets	20%
	WIT (CFI part) [3]	No: over-approximate valid sets	7%
	DFI [10]	No: over-approximate valid sets	104%
Non-Executable Data	HW (NX bit)	No: code reuse attacks	0%
	SW (Exec Shield, PaX)	No: code reuse attacks	few %
High-level policies	Sandboxing (SFI)	Isolation only	varies
	ACLs	Isolation only	varies
	Capabilities	Isolation only	varies

Figure 5: Summary of control-flow hijack defense mechanisms aligned with individual steps that are necessary for a successful attack. The figure on the left is a simplified version of the complete memory corruption diagram in [46].

Bound). We believe CPI is the first to stop all control-flow hijack attacks at this step.

Randomization techniques, like ASLR [40] and ASLP [26], mitigate attacks by restricting the attacker's knowledge of the memory layout of the application in Step 3. PointGuard [13] and DSR [7] (which is similar to probabilistic WIT) randomize the data representation by encrypting pointer values, but face compatibility problems. Software diversity [21] allows fine-grained, per-instance code randomization. Randomization techniques are defeated by information leaks through, e.g., memory corruption bugs [45] or side channel attacks [22].

Control-flow integrity [1] ensures that the targets of all indirect control-flow transfers point to valid code locations in Step 4. All CFI solutions rely on statically pre-computed context-insensitive sets of valid control-flow target locations. Many practical CFI solutions simply include every function in a program in the set of valid targets [53, 54, 29, 47]. Even if precise static analysis was be feasible, CFI could not guarantee protection against all control-flow hijack attacks, but rather merely restrict the sets of potential hijack targets. Indeed, recent results [19, 15, 9] show that many existing CFI solutions can be bypassed in a principled way. CFI+SFI [52], Strato [51] and MIPS [38] enforce an even more relaxed, statically defined CFI property in order to enforce software-based fault isolation (SFI). CCFI [31] encrypts code pointers in memory and provides security guarantees close to CPS. Data-flow based techniques like data-flow integrity (DFI) [10] or dynamic taint analysis (DTA) [42] can enforce that the used code pointer was not set by an unrelated instruction or to untrusted data, respectively. These techniques may miss some attacks or cause false positives, and have higher performance costs than CPI and CPS. Stack cookies, CFI, DFI, and DTA protect control-transfer instructions by detecting illegal modification of the code pointer whenever it is used, while CPI protects the load and store of a code pointer, thus preventing the corruption in the first place. CPI provides precise and provable security guarantees.

In Step 5, the execution of injected code is prevented by enforcing the non-executable (NX) data policy, but code-reuse attacks remain possible.

High level policies, e.g., restricting the allowed system calls of an application, limit the power of the attacker even in the presence of a successful control-flow hijack attack in Step 6. Software fault isolation (SFI) techniques [32, 18, 11, 50, 52] restrict indirect control-flow transfers and memory accesses to part of the address space, enforcing a sandbox that contains the attack. SFI prevents an attack from escaping the sandbox and allows the enforcement of a high-level policy, while CPI enforces the control-flow inside the application.

7 Conclusion

This paper describes *code-pointer integrity* (CPI), a way to protect systems against all control-flow hijacks that exploit memory bugs, and *code-pointer separation*, a relaxed form of CPI that still provides strong guarantees. The key idea is to selectively provide full memory safety for just a subset of a program's pointers, namely code pointers. We implemented our approach and showed that it is effective, efficient, and practical. Given its advantageous security-to-overhead ratio, we believe our approach marks a step toward deterministically secure systems that are fully immune to control-flow hijack attacks.

Acknowledgments

We thank the anonymous reviewers and our shepherd Junfeng Yang for their valuable input. We are grateful to Martin Abadi, Herbert Bos, Miguel Castro, Vijay D'Silva, Ulfar Erlingsson, Johannes Kinder, Per Larsen, Jim Larus, Santosh Nagarakatte, and Jonas Wagner for

$$
\begin{array}{rll}
\text{Atomic Types} & a & ::= \texttt{int} \mid p* \\
\text{Pointer Types} & p & ::= a \mid s \mid f \mid \texttt{void} \\
\text{Struct Types} & s & ::= \texttt{struct}\{\ldots; a_i : id_i; \ldots\} \\
\text{LHS Expressions} & lhs & ::= x \mid *lhs \mid lhs.id \mid lhs\text{->}id \\
\text{RHS Expressions} & rhs & ::= i \mid \&f \mid rhs + rhs \mid lhs \mid \&lhs \\
 & & \mid (a)\ rhs \mid \texttt{sizeof(p)} \mid \texttt{malloc}(rhs) \\
\text{Commands} & c & ::= c;c \mid lhs = rhs \mid f() \mid (*lhs)()
\end{array}
$$

Figure 6: The subset of C used in Appendix A; x denotes local statically typed variables, id – structure fields, i – integers, and f – functions from a pre-defined set.

their valuable feedback and discussions on earlier versions of the paper. This work was supported by ERC Starting Grant No. 278656, a Microsoft Research PhD fellowship, a gift from Google, DARPA award HR0011-12-2-005, NSF grants CNS-0831298 and CNS-1319137, and AFOSR FA9550-09-1-0539.

A Formal Model of CPI

This section presents a formal model and operational semantics of the CPI property and a sketch of its correctness proof. Due to the size and complexity of C/C++ specifications, we focus on a small subset of C that illustrates the most important features of CPI. Due to space limitations we focus on spatial memory safety. We build upon the formalization of spatial memory safety in SoftBound [34], reuse the same notation, and extend it to support applying spatial memory safety to a subset of memory locations. The formalism can be easily extended to provide temporal memory safety, directly applying the CETS [35] mechanism to the safe memory region of the model. Fig. 6 gives the syntax rules of the C subset we consider in this section. All valid programs must also pass type checking as specified by the C standard.

We define the runtime environment E of a program as a triple (S, M_u, M_s), where S maps variable identifiers to their respective atomic types and addresses, a regular memory M_u maps addresses to values (denoted as v and called regular values), and a safe memory M_s maps addresses to values with bounds information (denoted as $v_{(b,e)}$ and called safe values) or a special marker none. The bounds information specifies the lowest (b) and the highest (e) address of the corresponding memory object. M_u and M_s use the same addressing, but might contain distinct values for the same address. Some locations (e.g., of void$*$ type) can store either safe or regular value and are resolved to either M_s or M_u at runtime.

The runtime provides the usual set of memory operations for M_u and M_s, as summarized in Table 5. M_u models standard memory, whereas M_s stores values with bounds and has a special marker for "absent" locations, similarly to the memory in SoftBound's [34] formalization. We assume the memory operations follow the standard behavior of read/write/malloc operations in all other respects, e.g., read returns the value previously written to the same location, malloc allocates a region of memory that is disjoint with any other allocated region, etc..

Operation	Semantics
$\texttt{read}_u\ M_u\ l$	return $M_u[l]$
$\texttt{write}_u\ M_u\ l\ v$	set $M_u[l] = v$
$\texttt{read}_s\ M_s\ l$	return $M_s[l]$, if l is allocated; return none otherwise
$\texttt{write}_s\ M_s\ l\ v_{(b,e)}$	set $M_s[l] = v_{(b,e)}$, if l is allocated; do nothing otherwise
$\texttt{write}_s\ M_s\ l\ \texttt{none}$	set $M_s[l] = \texttt{none}$, if l is allocated; do nothing otherwise
$\texttt{malloc}\ E\ i$	allocate a memory object of size i in both $E.M_u$ and $E.M_s$ (at the same address); fail when out of memory

Table 5: Memory Operations in CPI

$$
\begin{array}{lll}
\texttt{sensitive} & \texttt{int} & ::= \mathit{false} \\
\texttt{sensitive} & \texttt{void} & ::= \mathit{true} \\
\texttt{sensitive} & f & ::= \mathit{true} \\
\texttt{sensitive} & p* & ::= \texttt{sensitive}\ p \\
\texttt{sensitive} & s & ::= \bigvee\limits_{i \in \mathit{fields\ of\ s}} \texttt{sensitive}\ a_i
\end{array}
$$

Figure 7: The decision criterion for protecting types in CPI

Enforcing the CPI property with low performance overhead requires placing most variables in M_u, while still ensuring that all pointers that require protection at runtime according to the CPI property are placed in M_s. In this formalization, we rely on type-based static analysis as defined by the sensitive criterion, shown on Fig. 7. We say a type p is sensitive iff sensitive $p =$ *true*. Setting sensitive to true for all types would make the CPI operational semantics equivalent to the one provided by SoftBound and would ensure full spatial memory safety of all memory operations in a program.

The classification provided by the sensitive criterion is static and only determines which operations in a program to instrument. Expressions of sensitive types could evaluate to both safe or regular values at runtime, whereas expressions of regular types always evaluate to regular values. In particular, according to Fig. 7, void$*$ is sensitive and, hence, in agreement with the C specification, values of that type can hold any pointer value at runtime, either safe or regular.

We extend the SoftBound definition of the result of an operation to differentiate between safe and regular values and left-hand-side locations:

$$\text{Results} \quad r ::= v_{(b,e)} \mid v \mid l_s \mid l_u \mid \texttt{OK} \mid \texttt{OutOfMem} \mid \texttt{Abort}$$

where $v_{(b,e)}$ and v are the safe (with bounds information) and, respectively, regular values that result from a right hand side expression, l_u and l_s are locations that result from a safe and regular left-hand-side expression, OK is a result of a successful command, and OutOfMem and Abort are error codes. We assume that all operational semantics rules of the language propagate these error codes up to the end of the program unchanged.

Using the above definitions, we now formalize the operational semantics of CPI through three classes of rules. The $(E, lhs) \Rightarrow_l l_s : a$ and $(E, lhs) \Rightarrow_l l_u : a$ rules specify how left hand side expressions are evaluated to a safe or regular locations, respectively. The $(E, rhs) \Rightarrow_r (v_{(b,e)}, E')$ and $(E, rhs) \Rightarrow_r (v, E')$ rules specify how right hand side expressions are evaluated to safe values with bounds or regular values, respectively, possibly modifying the environment through memory allocation (turning it from E to E'). Finally, the $(E, c) \Rightarrow_c (r, E')$ rules specify how commands are executed, possibly modifying the environment, where r can be either OK or an error code. We only present the rules that are most important for the CPI semantics, omitting rules that simply represent the standard semantics of the C language.

Bounds information is initially assigned when allocating a memory object or when taking a function's address (both operations always return safe values):

$$\frac{\texttt{address}(f) = l}{(E, \&f) \Rightarrow_r (l_{(l,l)})} \qquad \frac{\begin{array}{l}(E, rhs) = i\\ \texttt{malloc}\, E\ i = (l, E')\end{array}}{(E, \texttt{malloc}(i)) \Rightarrow_r (l_{(l,l+i)}, E')}$$

Taking the address of a variable from S if its type is sensitive is analogous. Structure field access operations either narrow bounds information accordingly, or strip it if the type of the accessed field is regular.

Type casting results in a safe value iff a safe value is cast to a sensitive type:

$$\frac{\begin{array}{l}\texttt{sensitive}\, a'\\ (E, rhs) \Rightarrow_l v_{(b,e)} : a\end{array}}{(E, (a')rhs) \Rightarrow_r (v_{(b,e)}, E)} \qquad \frac{\begin{array}{l}\neg\texttt{sensitive}\, a'\\ (E, rhs) \Rightarrow_l v_{(b,e)} : a\end{array}}{(E, (a')rhs) \Rightarrow_r (v, E)}$$

$$\frac{(E, rhs) \Rightarrow_l v : a}{(E, (a')rhs) \Rightarrow_r (v, E)}$$

The next set of rules describes memory operations (pointer dereference and assignment) on sensitive types and safe values:

$$\frac{\begin{array}{l}\texttt{sensitive}\, a\\ (E, lhs) \Rightarrow_l l_s : a*\\ \texttt{read}_s(E.M_s)l_s = \texttt{some}\ l'_{(b,e)}\\ l' \in [b, e - \texttt{sizeof}(a)]\end{array}}{(E, *lhs) \Rightarrow_l l'_s : a} \qquad \frac{\begin{array}{l}\texttt{sensitive}\, a\\ (E, lhs) \Rightarrow_l l_s : a*\\ \texttt{read}_s(E.M_s)l_s = \texttt{some}\ l'_{(b,e)}\\ l' \notin [b, e - \texttt{sizeof}(a)]\end{array}}{(E, *lhs) \Rightarrow_l \texttt{Abort}}$$

$$\frac{\begin{array}{l}\texttt{sensitive}\, a\\ (E, lhs) \Rightarrow_l l_s : a\\ (E, rhs) \Rightarrow_r v_{(b,e)} : a\\ E'.M_s = \texttt{write}_\texttt{s}(E.M_s) l_s\ v_{(b,e)}\end{array}}{(E, lhs = rhs) \Rightarrow_c (\texttt{OK}, E')}$$

These rules are identical to the corresponding rules of SoftBound [34] and ensure full spatial memory safety of all memory objects in the safe memory. Only operations matching those rules are allowed to access safe memory M_s. In particular, any attempts to access values of sensitive types through regular lvalues cause aborts:

$$\frac{\begin{array}{l}\texttt{sensitive}\, a\\ (E, lhs) \Rightarrow_l l_u : a*\end{array}}{(E, *lhs) \Rightarrow_l \texttt{Abort}} \quad \frac{\begin{array}{l}\texttt{sensitive}\, a\\ (E, lhs) \Rightarrow_l l_u : a\end{array}}{(E, lhs = rhs) \Rightarrow_c (\texttt{Abort}, E)}$$

Note that these rules can only be invoked if the value of the sensitive type was obtained by casting from a regular type using a corresponding type casting rule. Levee relaxes the casting rules to allow propagation of bounds information through certain right-hand-side expressions of regular types. This relaxation handles most common cases of unsafe type casting; it affects performance (inducing more instrumentation) but not correctness.

Some sensitive types (only `void*` in our simplified version of C), can hold regular values at runtime. For example, a variable of `void*` type can first be used to store a function pointer and subsequently re-used to store an `int*` value. The following rules handle such cases:

$$\frac{\begin{array}{l}\texttt{sensitive}\, a\\ (E, lhs) \Rightarrow_l l_s : a*\\ \texttt{read}_s(E.M_s)l = \texttt{none}\\ \texttt{read}_u(E.M_u)l = l'\end{array}}{(E, *lhs) \Rightarrow_l l'_u : a} \quad \frac{\begin{array}{l}\texttt{sensitive}\, a\\ (E, lhs) \Rightarrow_l l_s : a\\ (E, rhs) \Rightarrow_r v : a\\ E'.M_u = \texttt{write}_\texttt{u}(E.M_u)\ l\ v\\ E'.M_s = \texttt{write}_\texttt{s}(E.M_s)\ l\ \texttt{none}\end{array}}{(E, lhs = rhs) \Rightarrow_c (\texttt{OK}, E')}$$

Memory operations on regular types always access regular memory, without any additional runtime checks, following the unsafe memory semantics of C.

$$\frac{\begin{array}{l}\neg\texttt{sensitive}\, a\\ (E, lhs) \Rightarrow_l l : a*\\ \texttt{read}_u(E.M_u)l = l'\end{array}}{(E, *lhs) \Rightarrow_l l'_u : a} \quad \frac{\begin{array}{l}\neg\texttt{sensitive}\, a\\ (E, lhs) \Rightarrow_l l : a\\ (E, rhs) \Rightarrow_r v : a\\ E'.M_u = \texttt{write}_\texttt{u}(E.M_u)\ l\ v\end{array}}{(E, lhs = rhs) \Rightarrow_c (\texttt{OK}, E')}$$

These accesses to regular memory can go out of bounds but, given that $\texttt{read}_u$ and $\texttt{write}_u$ operations can only modify regular memory M_u, it does not violate memory safety of the safe memory.

Finally, indirect calls abort if the function pointer being called is not safe:

$$\frac{(E, lhs) \Rightarrow_r l_s : f*}{(E, (*lhs)()) \Rightarrow_c (\texttt{OK}, E')} \quad \frac{(E, lhs) \Rightarrow_r l_u : f*}{(E, (*lhs)()) \Rightarrow_c (\texttt{Abort}, E)}$$

Note that the operational rules for values that are safe at runtime are fully equivalent to the corresponding SoftBound rules [34] and, therefore, satisfy the SoftBound safety invariant which, as proven in [34], ensures memory safety for these values. According to the `sensitive` criterion and the safe location dereference and indirect function call rules above, all dereferences of pointers that require protection according to the CPI property are always safe at runtime, or the program aborts. Therefore, the operational semantics defined above indeed ensure the CPI property as defined in §3.1.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conf. on Computer and Communication Security*, 2005.

[2] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, 2010.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symp. on Security and Privacy*, May 2008.

[4] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-compatible Defense Against Out-of-bounds Errors. In *USENIX Security Symposium*, 2009.

[5] G. Altekar and I. Stoica. Focus replay debugging effort on the control plane. *USENIX Workshop on Hot Topics in Dependability*, 2010.

[6] S. Bhatkar, E. Bhatkar, R. Sekar, and D. C. Duvarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[7] S. Bhatkar and R. Sekar. Data Space Randomization. In *Intl. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[8] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Symp. on Information, Computer and Communications Security*, 2011.

[9] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[10] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Symp. on Operating Systems Design and Implementation*, 2006.

[11] 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 *ACM Symp. on Operating Systems Principles*, 2009.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conf. on Computer and Communication Security*, 2010.

[13] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointguardTM: protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, 2003.

[14] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[15] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[16] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[17] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: enforcing alias analysis for weakly typed languages. *SIGPLAN Notices*, 41(6):144–157, June 2006.

[18] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Symp. on Operating Systems Design and Implementation*, 2006.

[19] E. Göktaş, E. Athanasopoulosy, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symp. on Security and Privacy*, 2014.

[20] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *IEEE/ACM Symp. on Code Generation and Optimization*, 2012.

[21] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *IEEE/ACM Symp. on Code Generation and Optimization*, 2013.

[22] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE Symp. on Security and Privacy*, 2013.

[23] Intel Architecture Instruction Set Extensions Programming Reference. `http://download-software.intel.com/sites/default/files/319433-015.pdf`, 2013.

[24] Intel. Introduction to Intel memory protection extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, July 2013.

[25] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conf.*, 2002.

[26] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity softwar. In *Annual Computer Security Applications Conf.*, 2006.

[27] C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *ACM Conf. on Programming Language Design and Implementation*, 2005.

[28] C. Lattner, A. Lenharth, and V. Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *ACM Conf. on Programming Language Design and Implementation*, 2007.

[29] J. Li, Z. Wang, T. K. Bletsch, D. Srinivasan, M. C. Grace, and X. Jiang. Comprehensive and efficient protection of kernel control data. *IEEE Transactions on Information Forensics and Security*, 6(4):1404–1417, Dec. 2011.

[30] The LLVM compiler infrastructure. `http://llvm.org/`.

[31] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. Cryptographically enforced control flow integrity. http://arxiv.org/abs/1408.1451, Aug. 2014.

[32] S. McCamant and G. Morrisett. Evaluating sfi for a cisc architecture. In *USENIX Security Symposium*, 2006.

[33] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Intl. Symp. on Computer Architecture*, 2012.

[34] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Safety for C. In *ACM Conf. on Programming Language Design and Implementation*, 2009.

[35] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Intl. Symp. on Memory Management*, 2010.

[36] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. on Programming Languages and Systems*, 27(3):477–526, 2005.

[37] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 11(58):`http://phrack.com/issues.html?issue=67&id=8`, Nov. 2007.

[38] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *ACM Conf. on Computer and Communication Security*, 2013.

[39] B. Niu and G. Tan. Modular control-flow integrity. In *ACM Conf. on Programming Language Design and Implementation*, 2014.

[40] PaX-Team. PaX ASLR (Address Space Layout Randomization). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[41] Phoronix. Phoronix test suite. `http://www.phoronix-test-suite.com/`.

[42] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symp. on Security and Privacy*, 2010.

[43] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conf.*, 2012.

[44] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conf. on Computer and Communication Security*, 2007.

[45] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symp. on Security and Privacy*, pages 574–588, 2013.

[46] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. *IEEE Symp. on Security and Privacy*, 2013.

[47] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, 2014.

[48] A. van de Ven and I. Molnar. Exec Shield. `https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.

[49] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: Runtime intrusion prevention evaluator. In *Annual Computer Security Applications Conf.*, 2011.

[50] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symp. on Security and Privacy*, 2009.

[51] B. Zeng, G. Tan, and Ú. Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Security Symposium*, 2013.

[52] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *ACM Conf. on Computer and Communication Security*, 2011.

[53] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Symp. on Security and Privacy*, 2013.

[54] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.