Code-Pointer Integrity

Volodymyr Kuznetzov, László Szekeres, Mathias Payer,
George Candea, R. Sekar, Dawn Song

In this chapter, we describe 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 that exploit memory corruption errors, including attacks that bypass control-flow integrity mechanisms, such as control-flow bending [Carlini et al. 2015e]. We also describe 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, and they are practical (CPI and CPS were used to protect a complete FreeBSD system and over 100 packages like `apache` and `postgresql`), effective (prevented 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++.

This chapter is organized as follows: we introduce the motivation and key ideas behind CPI and CPS (Section 4.1), describe related work (Section 4.2), introduce our threat model (Section 4.3), describe CPI and CPS design (Section 4.4), present the formal model of CPI (Section 4.5), describe an implementation of CPI (Section 4.6) and the experimental results (Section 4.7), and then conclude (Section 4.8).

4.1 Introduction

System 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 [Wojtczuk 1998, Nergal 2001, Checkoway et al. 2010, Bletsch et al. 2011].

The goal of CPI is to secure system 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 CPI makes 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 mechanisms that can reduce the risk of control-flow hijack attacks without imposing undue overhead. Data Execution Prevention (DEP) and W ⊕ X [van de Ven 2004] use 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 [Nergal 2001] and return-oriented programming (ROP) [Wojtczuk 1998, Bletsch et al. 2011], which can construct arbitrary Turing-complete computations by chaining together existing code fragments of the original application. Address Space Layout Randomization (ASLR) [PaX Team 2004a] 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-channel attacks [Hund et al. 2013], and just-in-time code-reuse attacks [Snow et al. 2013]. Finally, stack cookies [Cowan et al. 1998] 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 [Szekeres et al. 2013], 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 [Jim et al. 2002, Necula et al. 2005], compiler modifications [Cowan et al. 2003, Akritidis et al. 2008, Dhurjati et al. 2006, Nagarakatte et al. 2009, Serebryany et al. 2012], or rewrite machine code binaries [Niu and Tan 2013, Zhang and Sekar 2013, Zhang et al. 2013].

Control-flow integrity (CFI) protection [Abadi et al. 2005a, Burow et al. 2016, Li et al. 2011, Zhang et al. 2013, Zhang and Sekar 2013, Niu and Tan 2014a], a widely studied technique for practical protection against control-flow hijack attacks, was recently demonstrated to be ineffective [Carlini et al. 2015e, Evans et al. 2015, Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014] against attacks that are adjusted to the constraints of the underlying defense.

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 $\geq 2\times$ overhead [Nagarakatte et al. 2010]. 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 chapter describes 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 describe 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 strong guarantees and incurs negligible overhead.

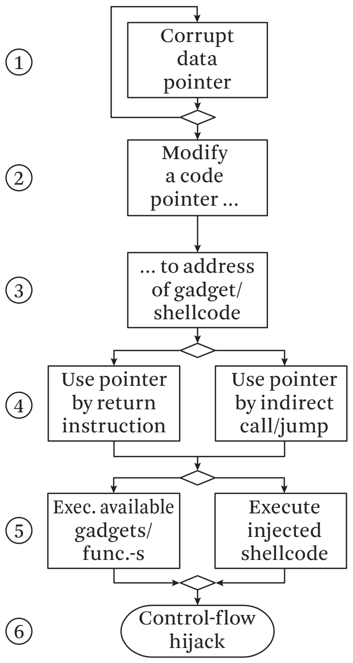
Finally, we describe SafeStack, a component of both CPI and CPS that protects code pointers on the stack. SafeStack is integrated into the Clang compiler starting with version 3.7.0.

The experimental evaluation of CPI and CPS shows that these techniques impose 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 prevented 100% of the attacks in the RIPE benchmark and the recent attacks [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014] that bypass CFI, ASLR, DEP, and all other Microsoft Windows protections. We compile and run with CPI/CPS a complete FreeBSD distribution along with over 100 widely used packages, demonstrating that the approach is practical.

4.2 Related Work

A variety of defense mechanisms have been proposed to date to answer the increasing challenge of control-flow hijack attacks, some of them described in Chapter 1. Figure 4.1 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 [Jim et al. 2002] and CCured [Necula et al. 2005] 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 [Nagarakatte et al. 2009] with its CETS [Nagarakatte et al. 2010] extension enforces *complete* memory safety at the cost of 2–4× slowdown. Tools with less overhead, like BBC [Akritidis et al. 2009], only *approximate* memory safety. LBC [Hasabnis et al. 2012] and Address Sanitizer [Serebryany et al. 2012] 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) [Akritidis et al. 2008] 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 [Dhurjati et al. 2006, Akritidis 2010] enforce type-safe memory reuse to mitigate attacks that exploit temporal errors (use-after-frees).



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

Attack step	Property	Mechanism	Mechanism	Average overhead
1	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%
2	**Code-pointer integrity**	CPI	**Yes**	8.4%
		CPS	No: valid code ptrs. interchangeable	1.9%
		SafeStack	No: precise return protection only	~0%
3	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%
4	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%
5	Non-executable data	HW (NX bit)	No: code-reuse attacks	0%
		SW (Exec Shield, PaX)	No: code-reuse attacks	few %
6	High-level policies	Sandboxing (SFI)	Isolation only	varies
		ACLs	Isolation only	varies
		Capabilities	Isolation only	varies

Figure 4.1 Summary of control-flow hijack defense mechanisms aligned with individual steps that are necessary for a successful attack. The diagram on the left is a simplified version of the complete memory corruption diagram in [Szekeres et al. 2013].

CPI by design enforces spatial and temporal memory safety for a subset of data (code pointers) in step 2 of Figure 4.1. Our Levee prototype currently enforces spatial memory safety and may be extended to enforce temporal memory safety as well (e.g., how CETS extends SoftBound). We believe CPI is the first to stop all control-flow hijack attacks at this step.

Randomization-based confidentiality techniques, like ASLR [PaX Team 2004a] and ASLP [Kil et al. 2006], mitigate attacks by restricting the attacker's knowledge of the memory layout of the application in step 3. PointGuard [Cowan et al. 2003] and DSR [Bhatkar and Sekar 2008] (which is similar to probabilistic WIT) randomize the data representation by encrypting pointer values but face compatibility problems. Software diversity [Homescu et al. 2013] allows fine-grained, per-instance code randomization. Randomization techniques are defeated by information leaks through, e.g., memory corruption bugs [Snow et al. 2013] or side-channel attacks [Hund et al. 2013].

Control-flow integrity [Abadi et al. 2005a] 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 [Zhang et al. 2013, Zhang and Sekar 2013, Li et al. 2011, Tice et al. 2014]. Even if precise static analysis would 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 [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014] show that many existing CFI solutions can be bypassed in a principled way. CFI+SFI [Zeng et al. 2011], Strato [Zeng et al. 2013a], and MIPS [Niu and Tan 2013] enforce an even more relaxed, statically defined CFI property in order to enforce software-based fault isolation. CCFI [Mashtizadeh et al. 2014] encrypts code pointers in memory and provides security guarantees close to CPS. Data-flow-based techniques, like Data-Flow Integrity (DFI) [Castro et al. 2006] or Dynamic Taint Analysis (DTA) [Schwartz et al. 2010], 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 [McCamant and Morrisett 2006, Erlingsson et al. 2006, Castro et al. 2009, Yee et al. 2009, Zeng et al. 2011] 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.

CPI and CPS rely on an instruction-level isolation mechanism to enforce the separation of code pointers from the rest of the data in program memory (Section 4.4). This book describes multiple implementations of such mechanisms (Section 4.6.3), including implementations that provide precise isolation guarantees (based on hardware- or software-enforced isolation) as well as probabilistic guarantees (based on randomization and information hiding). Evans et al. [2015a] demonstrated that one of the implementations with probabilistic guarantees can be bypassed in practical settings. Their attack cannot subvert the other implementation alternatives presented in this book (see Section 4.6.3 and [Kuznetsov et al. 2015]).

4.3 Threat Model

This chapter 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 shellcode, (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., those that modify or leak unprotected non-control data, are outside the scope of our discussion.

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.

4.4 Design

We now present the terminology used to describe our design, then define the code-pointer integrity property (Section 4.4.1), describe the corresponding enforcement mechanism (Section 4.4.2), and define a relaxed version that trades some security guarantees for performance (Section 4.4.4). We further formalize the CPI enforcement mechanism and sketch its correctness proof (Section 4.5).

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 either are themselves based on object X or are not pointers. This is a 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 [Nagarakatte et al. 2009, Nagarakatte et al. 2010]. Precise memory safety enforcement [Nagarakatte et al. 2009, Necula et al. 2005, Jim et al. 2002] 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.

4.4.1 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 Figure 4.2. 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 Section 4.4.4). The notion of a sensitive pointer

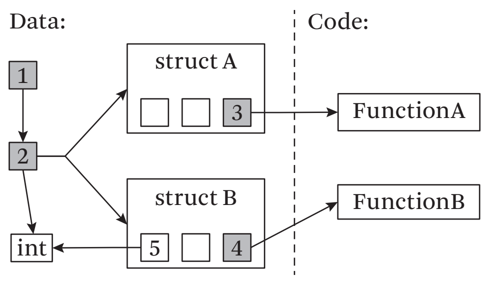


Figure 4.2 CPI protects code pointers 3 and 4 and pointers 1 and 2 (which may access pointers 3 and 4 indirectly). Pointer 2 of type `void*` may point to different objects at different times. The `int*` pointer 5 and non-pointer data locations are not protected.

is dynamic. For example, a `void*` pointer 2 in Figure 4.2 is sensitive when it points at another sensitive pointer at runtime, but it is not sensitive when it points to an integer.

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 [Nagarakatte et al. 2010]. Our observation is that only a small subset of all pointers is 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 [Altekar and Stoica 2010], 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 runtime. 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.

4.4.2 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 (Section 4.4.2.1), 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 (Section 4.4.2.2), and an instruction-level *isolation* mechanism that prevents non-protected memory operations from accessing the safe region (Section 4.4.2.3). For performance reasons, we handle return addresses stored on the stack separately from the rest of the code pointers using a *safe stack* mechanism (Section 4.4.3).

4.4.2.1 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 contain 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 call 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 [Nagarakatte et al. 2009].

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 Section 4.7, 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× 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 [Lattner and Adve 2005, Lattner et al. 2007] could prove more effective.

4.4.2.2 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 (Figure 4.3). 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 runtime 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 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 Figure 4.3). The layout of the safe pointer store is similar to metadata storage in SoftBounds+CETS [Nagarakatte et al. 2010],

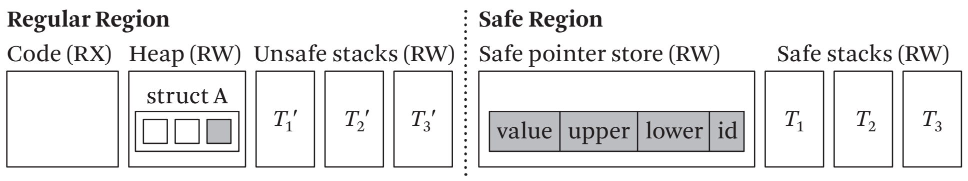


Figure 4.3 CPI memory layout: The safe region contains the safe pointer store and the safe stacks. The location of a sensitive pointer on the left (shaded) remains unused, while the value of this pointer and its metadata are stored in the safe pointer store. The safe stacks T_1, T_2, T_3 have corresponding stacks T_1', T_2', T_3' in regular memory to allocate unsafe stack objects.

except that CPI *also* stores the value of p in the safe pointer store. Combined with the isolation of the safe region (Section 4.4.2.3), this allows CPI to guarantee full memory safety of all sensitive pointers without having to instrument 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 Section 4.4.1. 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 describes 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 (Section 4.4.2.3) 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 pointer, and so would need to be instrumented, but we instead protect return addresses using a safe stack (Section 4.4.3).

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 the 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 (e.g., 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.

4.4.2.3 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 (Section 4.4.3). 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.

Certain architectures provide other hardware-enforced isolation mechanisms that can be used to protect the safe region. For instance, Intel recently introduced the memory protection keys extension and the MPX extension [Intel 2013] for the x86-64 architecture. These extensions could be used to isolate the safe region with low overhead, as discussed in Section 4.6.4.

On other architectures, CPI can protect the safe region using precise Software Fault Isolation (SFI) [Castro et al. 2009]. 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%.

On 64-bit architectures, CPI could also protect the safe region using randomization and information hiding. The fact that no addresses pointing into the safe region are ever stored in the regular region is what makes perfect information hiding possible. For example, the x86-64 architecture no longer enforces the segment limits; however, it still provides two segment registers with configurable base addresses. Similar 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. Hiding large regions through randomization is not secure [Oikonomopoulos et al. 2016]; however, when the size of the safe region is small (e.g., when the region is implemented as a hash table, as discussed in Section 4.6), the 48-bit address space of modern x86-64 CPUs makes guessing the safe region address impractical at least in some usage scenarios.

We further analyze the security and performance implications of the safe region protection mechanisms described above in Section 4.6.

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 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 Section 4.6.

4.4.3 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 runtime 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 4.4). 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 library. 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 [Bhatkar et al. 2005] and XFI [Erlingsson et al. 2006], 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 [Jim et al. 2002] or CCured [Necula et al. 2005] that simply allocate these objects on the heap, but our approach has significantly lower performance overhead.

Compared to a shadow stack like in CFI [Abadi et al. 2005a], 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; and (iii) the safe stack has near-zero performance overhead (Section 4.7.2), 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 [Cowan et al. 1998] 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.

4.4.4 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 and 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 (1) code pointers can only be stored to or modified in memory by code-pointer store instructions, and (2) code pointers can only be loaded by code-pointer load instructions from memory locations to which a code-pointer store instruction previously stored a value. Guarantee (1) restricts the attack surface, while guarantee (2) 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 during its execution. Combined with the safe stack, CPS precisely protects return addresses.

In contrast, CFI [Abadi et al. 2005a] allows any vulnerable instruction in a program to modify any code pointer; it only checks that the value of a code pointer, when used in an indirect control transfer, is within the set of allowed destinations, as defined by a specific implementation of CFI. For instance, coarse-grained CFI implementations [Zhang and Sekar 2013, Zhang et al. 2013] define allowed destinations as any function defined in a program (for function pointers) or that directly follows a call instruction (for return addresses). 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. Fine-grained 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.

For C++ programs, CPS protects not only code pointers but also virtual table pointers. The abundance of virtual table pointers in most C++ programs gives an attacker sufficient freedom to induce malicious program behavior by only chaining existing virtual functions through corresponding existing call sites [Schuster et al. 2015]. Including virtual table pointers in the set of sensitive pointers protected by CPS prevents such attacks.

CPS provides strong control-flow integrity guarantees and incurs low overhead (Section 4.7). We found that it prevents all recent attacks designed to bypass CFI [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014, Carlini et al. 2015e, Evans et al. 2015]. We consider CPS to be a solid alternative to CPI in those cases when CPI's (already low) overhead seems too high.

4.5 The 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 [Nagarakatte et al. 2009], 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

Atomic types	a	$::=$ `int` $\mid p*$
Pointer types	p	$::= a \mid s \mid f \mid$ `void`
Struct types	s	$::=$ `struct{...;` a_i`:`id_i`;...}`
LHS expressions	lhs	$::= x \mid *lhs \mid lhs.id \mid lhs$`->`$id$
RHS expressions	rhs	$::= i \mid \&f \mid rhs + rhs \mid lhs \mid \&lhs$ $\mid$ `(`a`)`$rhs \mid$ `sizeof(`p`)` $\mid$ `malloc(`rhs`)`
Commands	c	$::= c$`;` $c \mid lhs = rhs \mid f$`()` $\mid$ `(*` lhs`)()`

Figure 4.4 The subset of C. x denotes local statically typed variables; id denotes structure fields; i denotes integers; and f denotes functions from a pre-defined set.

`sensitive`	`int`	$::=$ *false*
`sensitive`	`void`	$::=$ *true*
`sensitive`	f	$::=$ *true*
`sensitive`	$p*$	$::=$ `sensitive` p
`sensitive`	s	$::= \bigvee_{i \in \text{fields of } s}$ `sensitive` a_i

Figure 4.5 The `sensitive` criterion for protecting types in CPI.

temporal memory safety, directly applying the CETS mechanism [Nagarakatte et al. 2010] to the safe memory region of the model. Figure 4.4 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 v and called regular values), and a safe memory M_s maps addresses to values with bounds information (denoted $v_{(b,e)}$ 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 4.1. M_u models standard memory, whereas M_s stores values

Table 4.1 **Memory Operations in CPI**

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

with bounds and has a special marker for "absent" locations, similarly to the memory in SoftBound's [Nagarakatte et al. 2009] 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 and malloc allocates a region of memory that is disjoint with any other allocated region.

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 in Figure 4.5. We say a type p is sensitive iff $\texttt{sensitive}\ p = \mathit{true}$. 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 Figure 4.5, 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 unchanged up to the end of the program.

Using the above definitions, we now formalize the operational semantics of CPI through three classes of rules. The $(E, \mathit{lhs}) \Rightarrow_l l_s : a$ and $(E, \mathit{lhs}) \Rightarrow_l l_u : a$ rules specify how left-hand-side expressions are evaluated to safe or regular locations, respectively. The $(E, \mathit{rhs}) \Rightarrow_r (v_{(b,e)}, E')$ and $(E, \mathit{rhs}) \Rightarrow_r (v, E')$ 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):

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

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:

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

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

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

These rules are identical to the corresponding rules of SoftBound [Nagarakatte et al. 2009] 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:

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

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 reused to store an `int*` value. The following rules handle such cases:

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

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

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

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, they do not violate memory safety of the safe memory.

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

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

Note that the operational rules for values that are safe at runtime are fully equivalent to the corresponding SoftBound rules [Nagarakatte et al. 2009]: the rules expressions are equal assuming `sensitive` a is `true` and, depending on the rule, either $\texttt{read}_\texttt{s}$ is not `none` or the right-hand-side value is sensitive. Therefore, under these conditions, these rules satisfy the SoftBound safety invariant, which, as proven in Nagarakatte et al. [2009], ensures memory safety for such 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 Section 4.4.1.

4.6 Implementation

We describe a CPI/CPS enforcement tool for C/C++, called Levee, that was implemented on top of the LLVM 3.3 compiler infrastructure [LLUM 2017], 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. The SafeStack component of Levee is already integrated upstream into the Clang compiler [SafeStack 2017], and there is an ongoing effort to upstream the rest of CPI/CPS in the future as well.

4.6.1 Analysis and Instrumentation Passes

CPI and CPS Instrumentation Passes. We implemented the static analysis and instrumentation for CPI as two LLVM passes, directly following the design from Section 4.4.4. 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 Section 4.4.4. The IR also does not distinguish between void* and char* (it 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.

4.6.2 Runtime support library

Most of the instrumentation by the passes discussed in Section 4.6.1 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

Table 4.2 **Security Guarantees and Performance Overhead of Various Implementations of CPI/CPS**

	Security[a]		Overhead[b]	
	CPI	CPS	CPI	CPS
Hardware segmentation	precise		8.4%	1.9%
Software fault isolation	precise		13.8%	7.0%
Information hiding				
hash table	16.6	20.7	9.7%	2.2%
lookup table	15	17	8.9%	2.0%
simple table	5	7	8.4%	1.9%

a. Either precise or number of entropy bits.
b. Average on SPEC2006.

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 hash table. 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 start-up) and additional TLB pressure. Switching to superpages (2 MB on Linux) made this simple table the fastest implementation of the three. Note that due to the large virtual size of the simple table, the implementation based on it cannot be used in conjunction with randomization-based safe region isolation.

4.6.3 Safe Region Isolation

We implemented multiple mechanisms that efficiently enforce instruction-level isolation as required to protect the safe memory region: using hardware-enforced segmentation, software fault isolation, or randomization and information hiding. The security guarantees and performance implications of these mechanisms are summarized in Table 4.2; we discuss them in detail below. We focus on design choices behind each mechanism and ignore potential non-design bugs in the prototypes we released.

Hardware-Enforced Segmentation-Based Implementation. On architectures that support hardware-enforced segmentation (e.g., x86-32 and some x86-64), CPI uses this feature directly to enforce instruction-level isolation. In such implementations, CPI dedicates a segment register to point to the safe memory region, and it enforces, at compile time, that only instructions instrumented with memory safety checks use this segment register. CPI configures all other segment registers, which are used by non-instrumented instructions, to prevent all accesses to the safe region through these segment registers on the hardware level.

Hardware-enforced segmentation is supported on x86-32 CPUs but also on some x86-64 CPUs (see the Long Mode Segment Limit Enable flag), which demonstrates that adding segmentation to x86-64 CPUs is feasible, provided the techniques that could benefit from it indeed prove to be valuable.

This implementation of CPI is precise and imposes zero performance overhead on instructions that do not access sensitive pointers.

Software Fault Isolation–Based Implementation. On architectures where hardware-enforced segmentation is not available (e.g., ARM and most of the x86-64 CPUs), the instruction-level isolation can be enforced using lightweight software fault isolation (SFI). In our implementation, we align the safe region in memory so that enforcing a pointer not to alias with it can be done with a single bitmask operation (unlike more heavyweight SFI solutions, which typically add extra memory accesses and/or branches for each memory access in a program). Furthermore, accesses to the safe stack need not be instrumented, as they are guaranteed to be safe [Kuznetsov et al. 2014a].

Our SFI-based implementation of CPI is precise, and SFI increases the overhead by less than 5% relative to hardware-enforced segmentation.

Information-Hiding-Based Implementation. Another way to implement instruction-level isolation is based on randomization and information hiding. Such implementations exploit the guarantee of the CPI instrumentation that, in a CPI-instrumented program, no pointers into the safe region are ever stored outside the safe region itself. When the base location of the safe region is randomized, the above guarantee implies that the attacker has to resort to random guessing in order to find the safe region, even in the presence of an arbitrary memory read vulnerability. On 64-bit architectures, most of the address space is unmapped and so most of the failed guesses result in a crash. Such crashes, if frequent enough, can be detected by other means.

The actual expected number of crashes required to find the location of the safe region by random guessing is determined by the size of the safe region and the size of the address space. Today's mainstream x86-64 CPUs provide 2^{48} bytes of address space (while the architecture itself envisions future extensions up to 2^{64} bytes). Half of the address space is usually occupied by the OS kernel, which leaves 2^{47} bytes for applications.

As explained above, the safe region stores a map that, for each sensitive pointer, maps the location that the pointer would occupy in the memory of a non-instrumented program to a tuple of the pointer value and its metadata. On 64-bit CPUs, each entry of this map occupies 32 bytes and, due to pointer alignment requirements, represents 8 bytes of program memory. The expected number of entries depends on the program memory usage, the fraction of sensitive pointers, and the data structure that is used to store this map.

We released three versions of our information-hiding-based CPI implementation that use either a hash table, a two-level lookup table, or a simple table to organize the safe region [Kuznetsov et al. 2014a]. Although all these safe region organizations are compatible with hardware-enforced segmentation and software-based fault isolation implementations, the choice of the organization has the highest impact on the information-hiding-based implementation. We analyze this impact. We estimate the size of the safe region and the expected number of crashes required to find its location for each of the versions below. For the purpose of this estimation, we assume a program uses 1GB of memory, 8% of which stores sensitive pointers (consistent with the experimental evaluation in [Kuznetsov et al. 2014a]), which amounts to $1\text{ GB} \times 8\%/8\text{ bytes} \approx 2^{23.4}$ sensitive pointers in total.

Hash table. This implementation is based on a linearly probed lookup table with a bitmask-and-shift-based hash function, which, due to sparsity of sensitive pointers in program memory, performs well with load factors of up to 0.5. Conservatively assuming a load factor of 0.25, the hash table would occupy $2^{23.4}/0.25 \times 32 = 2^{30.4}$ bytes of memory. Randomizing the hash table location can provide up to $47 - 30.4 = 16.6$ bits of entropy, requiring $2^{15.6} \approx$ 51,000 crashes on average to guess it. In most systems, that many crashes can be detected externally, making the attack infeasible.

Two-level lookup table. This implementation organizes the safe region similar to page tables, using the higher 23 bits of the address as an index in the directory and the lower 22 bits as an index in a subtable (the lowest 3 bits are zero due to alignment). Each subtable takes 32×2^{22} bytes and describes an 8×2^{22}-byte region of the address space. Assuming sensitive pointers are uniformly distributed across the 1GB of continuous program memory, CPI will allocate $1\text{ GB}/(8 \times 2^{22}) \times 32 \times 2^{22} = 2^{32}$ bytes for the subtables. Randomizing the subtable locations gives $47 - 32 = 15$ bits of entropy, requiring 2^{14} crashes on average to guess. Note that the attacker will find a random one among multiple subtables, and finding usable code pointers in it requires further guessing. This attack is thus also infeasible in many practical cases.

Simple table. This simple implementation allocates a fixed-size region of 2^{42} bytes for the safe region that maps addresses linearly. This implementation would give only $47 - 42 = 5$ bits of entropy. The location of the simple table in this implementation can be guessed while causing only 16 crashes on average. This level of protection is not sufficient for most practical cases.

Code-Pointer Separation, unlike full CPI, does not require any metadata and has fewer sensitive pointers (by 8.5× on average [Kuznetsov et al. 2014a]). This increases the number of expected crashes by 17× for the hash table-based implementation, and by 4× for the other two implementations.

The information-hiding-based implementation of CPI that uses a hash table to organize the safe region provides probabilistic security guarantees with $2^{16.6}$ bits of entropy (or $2^{20.7}$ for CPS). We believe that, in certain practical use cases, this number of crashes, especially given the uniform pattern of these crashes, can be detected automatically by external means.

In our evaluations, all three versions of information-hiding-based implementation have performance overhead comparable to the hardware-enforced segmentation.

4.6.4 Discussion

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 Section 4.4 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 [Nagarakatte et al. 2010] 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 Section 4.4.2 with data-based points-to analysis, such as DSA [Lattner and Adve 2005, Lattner et al. 2007].

Future MPX-Based Implementation. Intel recently introduced a hardware extension, Intel MPX, to be used for hardware-enforced memory safety [Intel 2013].

We believe MPX (or similar) hardware can be re-purposed 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 [Devietti et al. 2008] or Watchdog [Nagarakatte et al. 2012] 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.

4.7 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 (Section 4.7.1). 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 (Section 4.7.2). To demonstrate practicality, we recompile with CPI/CPS/safe stack the base FreeBSD plus over 100 packages and report results on several benchmarks (Section 4.7.3).

We ran our experiments on an Intel Xeon E5-2697 with 24 cores running at 2.7 GHz in 64-bit mode with 512 GB 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.

4.7.1 Effectiveness on the RIPE Benchmark

We described in Section 4.4 the security guarantees provided by CPI, CPS, and the safe stack based on their design; to experimentally evaluate their effectiveness, we use the RIPE [Wilander et al. 2011] 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 runtime 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 Section 4.4 and according to the proof outlined in Section 4.5, CPI renders all (known and unknown) memory corruption-based control-flow hijack attacks impossible.

4.7.2 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 4.3 and presented in detail in Figure 4.6. We also compare Levee to two related approaches, SoftBound [Nagarakatte et al. 2009] and control-flow integrity [Abadi et al. 2005a, Zhang and Sekar 2013, Zhang et al. 2013, Niu and Tan 2014a].

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

The next most important sources of overhead are `libc` memory manipulation functions, like `memset` and `memcpy`. When our static analysis cannot prove that a

Table 4.3 Summary of SPEC CPU2006 Performance Overheads

	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%

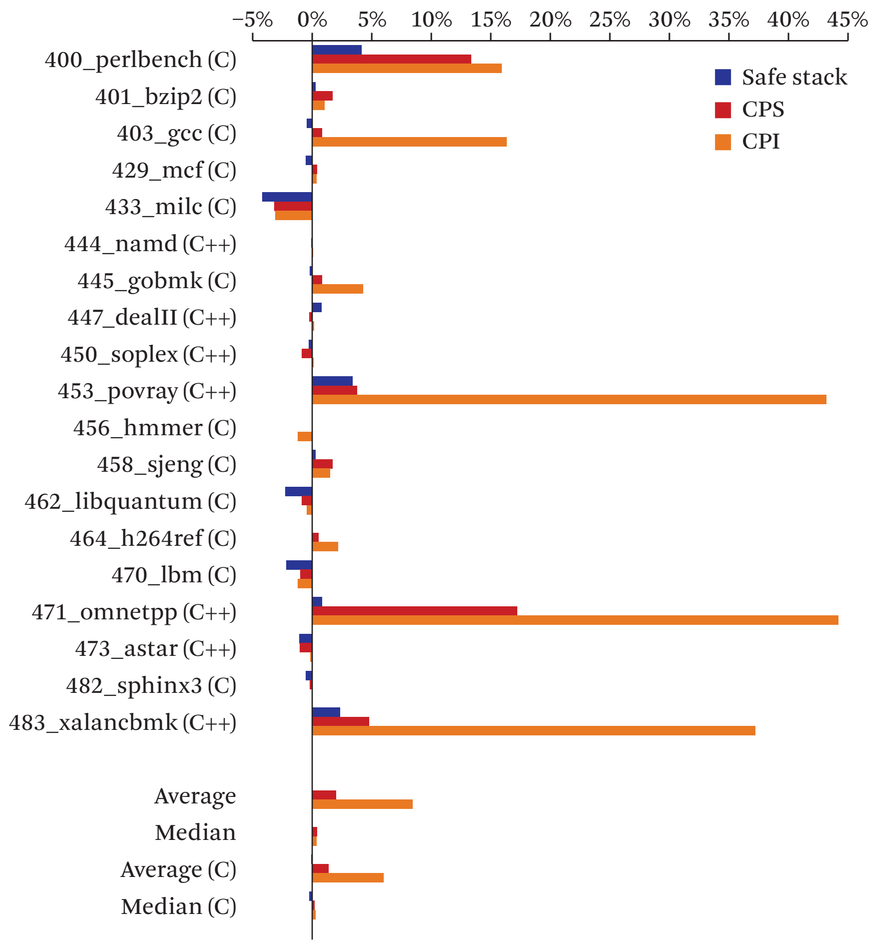


Figure 4.6 Levee performance for SPEC CPU2006, under three configurations: full CPI, CPS only, and safe stack only.

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 runtime, 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`, `xalancbmk`, 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 and believe it can be improved in future prototypes.

In Table 4.4 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 Section 4.4.3. 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 Figure 4.6.

Comparison to SoftBound. We compare with SoftBound [Nagarakatte et al. 2009] on the SPEC benchmarks. We cannot fairly reuse the numbers from that paper because

Table 4.4 Compilation Statistics for Levee

Benchmark	FN_{UStack} [a]	MO_{CPS} [b]	MO_{CPI} [b]
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%

a. FN_{UStack} lists what fraction of functions need an unsafe stack frame.
b. MO_{CPS} and MO_{CPI} show the fraction of memory operations instrumented for CPS and CPI, respectively.

they are based on an older version of SPEC. In Table 4.5 we report numbers for the four C/C++ SPEC benchmarks that can compile with the current version of Soft-Bound. 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 far fewer pointers. Many of the SPEC benchmarks either do not compile or terminate with an error when instrumented by SoftBound, which illustrates the practical impact of this difference.

Table 4.5 Overhead of Levee and SoftBound on SPEC Programs That Compile and Run Free of Errors with SoftBound

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%

Comparison to Control-Flow Integrity (CFI). The average overhead for compiler-enforced CFI is 21% for a subset of the SPEC CPU2000 benchmarks [Abadi et al. 2005a] and 5–6% for MCFI [Niu and Tan 2014a] (without stack pointer integrity). CCFIR [Zhang et al. 2013] reports an overhead of 3.6%, and binCFI [Zhang and Sekar 2013] reports 8.54% for SPEC CPU2006 to enforce a weak CFI property with globally merged target sets. WIT [Akritidis et al. 2008], a source-based mechanism that enforces both CFI and write integrity protection, has 10% overhead.

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 exist successful attacks against CFI [Göktas et al. 2014a, Davi et al. 2014, Carlini and Wagner 2014]. 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 Section 4.2.

4.7.3 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`, and `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 [Phoronix 2017], a widely used comprehensive benchmarking platform for operat-

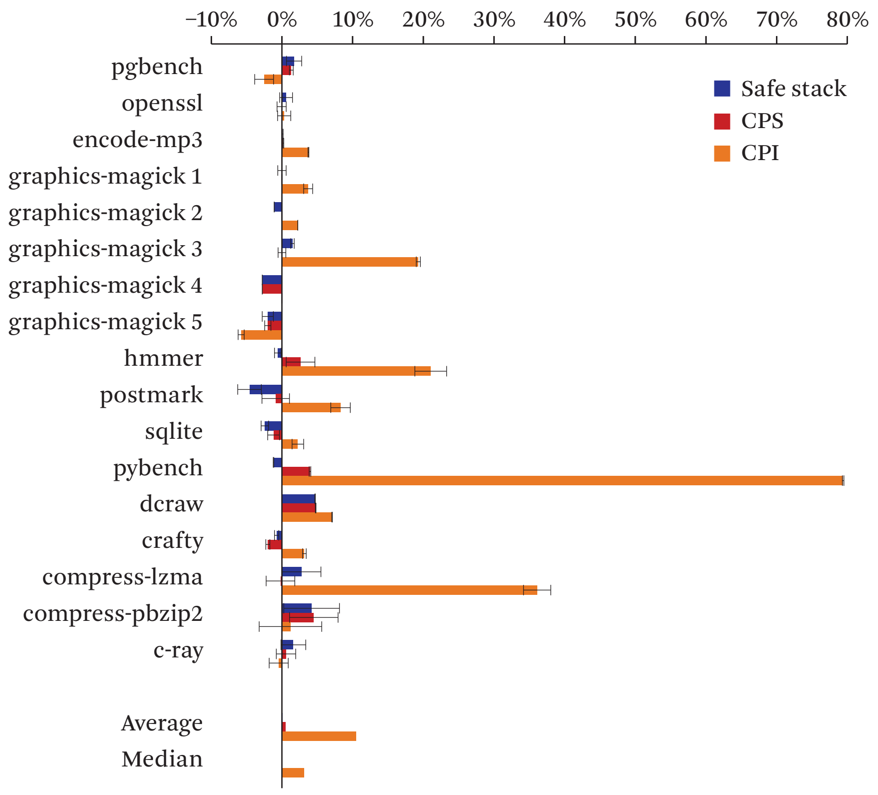


Figure 4.7 Performance overhead on FreeBSD (Phoronix).

ing 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.

Figure 4.7 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 Section 4.7.2. 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 Figure 4.7 shows, for many benchmarks the overhead of CPS and the safe stack are within the measurement error.

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

Table 4.6 **Throughput Benchmark for Web Server Stack (FreeBSD + Apache + SQLite + mod_wsgi + Python + Django)**

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%

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 4.6.

The CPI overhead for a dynamic page generated by Python code is much larger than we expected but consistent with suspiciously high overhead of the `pybench` benchmark in Figure 4.7. 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.

4.8 Conclusion

This chapter describes *code-pointer integrity* (CPI), a way to protect systems against all control-flow hijacks that exploit memory bugs, and *code-pointer separation* (CPS), 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. The CPI/CPS implementation and evaluation shows that it is effective, efficient, and practical. Given its advantageous security-to-overhead ratio, the authors believe this approach marks a step toward deterministically secure systems that are fully immune to control-flow hijack attacks.

References

M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2005a. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pp. 340–353. DOI: 10.1145/1102120.1102165. 12, 25, 38, 39, 62, 82, 86, 95, 97, 110, 114, 117, 139, 141, 173, 174, 181, 186, 211, 233, 243, 249

M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2005b. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Methods and Software Engineering (ICFEM)*. DOI: 10.1007/11576280_9. 182, 186

M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. 2009. Control-flow integrity: Principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1). DOI: 10.1145/1609956.1609960. 181, 189, 208

A. Acharya and M. Raje. 2000. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium (SSYM)*, pp. 1–17. 16

P. Akritidis. 2010. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pp. 177–192. 84, 173, 178

P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. 2008. Preventing memory error exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, pp. 263–277. DOI: 10.1109/SP.2008.30. 8, 58, 82, 84, 114, 173, 176, 178

P. Akritidis, M. Costa, M. Castro, and S. Hand. 2009. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pp. 51–66. 84, 173, 178

Aleph One. 1996. Smashing the stack for fun and profit. *Phrack*, 7. 11, 17

A. Alexandrov, P. Kmiec, and K. Schauser. 1999. Consh: Confined execution environment for Internet computations. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.488. DOI: 10.1.1.57.488. 16

G. Altekar and I. Stoica. 2010. Focus replay debugging effort on the control plane. In *USENIX Workshop on Hot Topics in Dependability*. 89

S. Andersen and V. Abella. August 2004. Changes to functionality in Windows XP service pack 2—part 3: Memory protection technologies. http://technet.microsoft.com/en-us/library/bb457155.aspx. 9, 19, 184

J. Ansel. March 2014. Personal communication. 53

J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. Schuff, D. Sehr, C. Biffle, and B. Yee. 2011. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 355–366. DOI: 10.1145/1993316.1993540. 58, 59

O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. 2015. HAFIX: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Design Automation Conference (DAC)*, pp. 74:1–74:6. DOI: 10.1145/2744769.2744847. 182, 208, 209

J.-P. Aumasson and D. J. Bernstein. 2012 SipHash: A fast short-input PRF. In *13th International Conference on Cryptology in India (INDOCRYPT)*. 73

M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security (CCS)*. DOI: 10.1145/2660267.2660378, pp. 1342–1353. 65, 173, 177

M. Backes and S. Nürnberger. 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium*, pp. 433–447. 64, 66

A. Balasubramanian, M. S. Baranowski, A. Burtsev, and A. Panda. 2017. System programming in Rust: Beyond safety. In *Workshop on Hot Topics in Operating Systems (HotOS)*, pp. 94–99. DOI: 10.1145/3102980.3103006. 79

C. Basile, Z. Kalbarczyk, and R. Iyer. 2002. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 149–158. DOI: 10.1109/DSN.2003.1209926. 230

C. Basile, Z. Kalbarczyk, and R. K. Iyer. 2006. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 17(5):448–465. DOI: 10.1109/TPDS.2006.56. 230

A. Basu, J. Bobba, and M. D. Hill. 2011. Karma: Scalable deterministic record-replay. In *Proceedings of the International Conference on Supercomputing*, pp. 359–368. DOI: 10.1145/1995896.1995950. 230

M. Bauer. 2006. Paranoid penguin: an introduction to Novell AppArmor. *Linux J.*, (148):13. 16

A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 335–348. 213

T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. 2010. CoreDet: A compiler and runtime system for deterministic multithreaded execution. *ACM SIGARCH Computer Architecture News*, 38(1):53–64. DOI: 10.1145/1735971.1736029. 230

E. Berger, T. Yang, T. Liu, and G. Novark. 2009. Grace: Safe multithreaded programming for C/C++. *ACM Sigplan Notices*, 44(10):81–96. 230

E. D. Berger and B. G. Zorn. 2006. DieHard: Probabilistic memory safety for unsafe languages. *ACM SIGPLAN Notices*, (6):158–168. DOI: 10.1145/1133255.1134000. 214

E. Bhatkar, D. C. Duvarney, and R. Sekar. 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the USENIX Security Symposium (SSYM)*, pp. 105–120. 10

S. Bhatkar and R. Sekar. 2008. Data space randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 1–22. DOI: 10.1007/978-3-540-70542-0_1. 11, 85

S. Bhatkar, R. Sekar, and D. C. DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium (SSYM)*, pp. 17–17. http://dl.acm.org/citation.cfm?id=1251398.1251415. 10, 95

D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 268–279. DOI: 10.1145/2810103.2813691. 11

A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. 2014. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pp. 227–242. DOI: 10.1109/SP.2014.22. 62, 140, 141, 182, 239

D. Blazakis. 2010. Interpreter exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, pp. 1–9. 59

T. Bletsch, X. Jiang, and V. Freeh. 2011. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 353–362. DOI: 10.1145/2076732.2076783. 208, 209

T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. 2011. Jump-oriented programming: A new class of code-reuse attack. In *Procceedings of the 6th ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 30–40. DOI: 10.1145/1966913.1966919. 20, 81, 82, 117

E. Bosman and H. Bos. 2014. Framing signals—a return to portable shellcode. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 243–258. DOI: 10.1109/SP.2014.23. 31, 140

K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. 2016. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*. 68

S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina. 2011. Exploit programming: From buffer overflows to "weird machines" and theory of computation. Usenix ;login: issue: December 2011, volume 36, number 6. 19

E. Buchanan, R. Roemer, H. Shacham, and S. Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pp. 27–38. DOI: 10.1145/1455770.1455776. 233

M. Budiu, Ú. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pp. 42–51. DOI: 10.1145/1181309.1181316. 208, 209

N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. 2016. Control-flow integrity: Precision, security, and performance. *Computing Research Repository (CoRR)*. 50(1). http://arxiv.org/abs/1602.04056. 12, 28, 62, 82

N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. 2017. Control-flow integrity: precision, security, and performance. *ACM Computing Surveys*. DOI: 10.1145/3054924. 12

J. Butler and anonymous. 2004. Bypassing 3rd party Windows buffer overflow protection. *Phrack*, 11. 17

N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium*, pp. 161–176. http://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini. 15, 20, 21, 59, 81, 82, 97, 137, 182, 183, 185, 186, 188, 200, 204, 211

N. Carlini and D. Wagner. 2014. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, pp. 385–399. http://dl.acm.org/citation.cfm?id=2671225.2671250. 15, 53, 82, 84, 86, 97, 114, 137, 139, 140, 176, 177, 179, 182, 183, 184, 186, 188, 200, 202, 209

M. Castro, M. Costa, and T. Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 147–160. 11, 86

M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. 2009. Fast byte-granularity software fault isolation. In *ACM Symposium on Operating Systems Principles*, pp. 45–58. DOI: 10.1145/1629575.1629581. 86, 93

L. Cavallaro. 2007. Comprehensive memory error protection via diversity and taint-tracking. PhD thesis, Universita Degli Studi Di Milano. 214, 223, 237

S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pp. 559–572. DOI: 10.1145/1866307.1866370. 20, 81, 117, 183, 184, 185, 186, 200, 202

S. Checkoway and H. Shacham. 2010. Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Technical report CS2010-0954, UC San Diego. http://cseweb.ucsd.edu/~hovav/dist/noret.pdf. 200, 202

P. Chen, Y. Fang, B. Mao, and L. Xie. 2011. JITDefender: A defense against JIT spraying attacks. In *26th IFIP International Information Security Conference*, volume 354, pp. 142–153. 60

P. Chen, R. Wu, and B. Mao. 2013. JITSafe: A framework against just-in-time spraying attacks. *IET Information Security*, 7(4):283–292. DOI: 10.1049/iet-ifs.2012.0142. 59, 60

S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. 2005 Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*. http://dl.acm.org/citation.cfm?id=1251398.1251410. 21, 184

X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. 2015. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Symposium on Network and Distributed System Security (NDSS)*. 173, 178

Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen. 2017. NORAX: Enabling execute-only memory for COTS binaries on AArch64. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 304–319. DOI: 10.1109/SP.2017.30. 68

Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21st Synposium on Network and Distributed System Security (NDSS)*. 117, 118, 119, 127, 173, 176, 182, 209

M. Co, J. W. Davidson, J. D. Hiser, J. C. Knight, A. Nguyen-Tuong, W. Weimer, J. Burket, G. L. Frazier, T. M. Frazier, and B. Dutertre, et al. 2016. Double Helix and RAVEN: A system for cyber fault tolerance and recovery. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, p. 17. DOI: 10.1145/2897795.2897805. 214

F. B. Cohen. 1993. Operating system protection through program evolution. *Computers & Security*, 12(6): 565–584. DOI: 10.1016/0167-4048(93)90054-9. 62

Corelan. 2011. Mona: A debugger plugin/exploit development Swiss army knife. http://redmine.corelan.be/projects/mona. 136

C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. 2000. SubDomain: Parsimonious server security. In *Proccedings of the 14th USENIX Conference on System Administration*, pp. 355–368. 16

C. Cowan, S. Beattie, J. Johansen, and P. Wagle. 2003. Pointguard™: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium (SSYM)*, pp. 7–7. http://dl.acm.org/citation.cfm?id=1251353.1251360. 11, 63, 76, 82, 85

C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 81, pp. 346–355. 61, 63, 82, 95, 211, 233

B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. 2006. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, 9. 211, 213, 214, 217, 237

S. Crane, A. Homescu, and P. Larsen. 2016. Code randomization: Haven't we solved this problem yet? In *IEEE Cybersecurity Development (SecDev)*. DOI: 10.1109/SecDev.2016.036. 66

S. Crane, P. Larsen, S. Brunthaler, and M. Franz. 2013. Booby trapping software. In *New Security Paradigms Workshop (NSPW)*, pp. 95–106. DOI: 10.1145/2535813.2535824. 68

S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy (S&P)*, pp. 763–780. DOI: 10.1109/SP.2015.52. 11, 60, 66, 76, 173, 178

S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. 2015. It's a TRaP: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 243–255. DOI: 10.1145/2810103.2813682. 11, 68, 77, 159, 171, 173, 178

J. Criswell, N. Dautenhahn, and V. Adve. 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 292–307. DOI: 10.1109/SP.2014.26. 58

H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. 2013. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 388–405. DOI: 10.1145/2517349.2522735. 230

D. Dai Zovi. 2010. Practical return-oriented programming. Talk at *SOURCE Boston*, 2010. 117

L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. 2010. Transactional mutex locks. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, pp. 2–13. 44

T. H. Y. Dang, P. Maniatis, and D. Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 555–566. DOI: 10.1145/2714576.2714635. 10, 137, 208

DarkReading. November 2009. Heap spraying: Attackers' latest weapon of choice. http://www.darkreading.com/security/vulnerabilities/showArticle.jhtml?articleID=221901428. 133

L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi. 2012. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*. 58, 208

L. Davi, P. Koeberl, and A.-R. Sadeghi. 2014. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Annual Design Automation Conference—Special Session: Trusted Mobile Embedded Computing (DAC)*, pp. 1–6. DOI: 10.1145/2593069.2596656. 173, 174, 209

L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, pp. 401–416. http://dl.acm.org/citation.cfm?id=2671225.2671251. 15, 43, 53, 82, 84, 86, 97, 114, 139, 140, 169, 174, 176, 177, 179, 182, 183, 184, 186, 188, 200, 209, 211

L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. 2015. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*. DOI: 10.14722/ndss.2015.23262. 64

L. Davi, A.-R. Sadeghi, and M. Winandy. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 40–51. DOI: 10.1145/1966913.1966920. 139, 141

L. de Moura and N. Bjørner. 2009. Generalized, efficient array decision procedures. In *Formal Methods in Computer Aided Design (FMCAD)*. DOI: 10.1109/FMCAD.2009.5351142. 161

L. M. de Moura and N. Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 337–340. 140, 161

T. de Raadt. 2005. Exploit mitigation techniques. http://www.openbsd.org/papers/ven05-deraadt/index.html. 8

J. Dean, D. Grove, and C. Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, pp. 77–101. 32

D. Dechev. 2011. The ABA problem in multicore data structures with collaborating operations. In *7th International Conference on Collaborative Computing: Networking, Applications, and Worksharing (CollaborateCom)*, pp. 158–167. DOI: 10.4108/icst.collaboratecom.2011.247161. 44

L. Deng, Q. Zeng, and Y. Liu. 2015. ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *30th International Conference on ICT Systems Security and Privacy Protection*, pp. 386–400. 41

L. P. Deutsch and A. M. Schiffman. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 297–302. DOI: 10.1145/800017.800542. 54

J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. 2008. HardBound: Architectural support for spatial safety of the C programming language. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 103–114. DOI: 10.1145/1353534.1346295. 109

J. Devietti, B. Lucia, L. Ceze, and M. Oskin. 2009. DMP: Deterministic shared memory multiprocessing. *ACM SIGARCH Computer Architecture News*, 37(1):85–96. DOI: 10.1145/1508244.1508255. 230

D. Dewey and J. T. Giffin. 2012. Static detection of C++ vtable escape vulnerabilities in binary code. In *Symposium on Network and Distributed System Security (NDSS)*. 171

D. Dhurjati, S. Kowshik, and V. Adve. June 2006. SAFECode: Enforcing alias analysis for weakly typed languages. *SIGPLAN Notices*, 41 (6): 144–157. DOI: 10.1145/1133255.1133999. 82, 84

U. Drepper. April 2006. SELinux memory protection tests. http://www.akkadia.org/drepper/selinux-mem.html. 238

V. D'Silva, M. Payer, and D. Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *LangSec'15: Second Workshop on Language-Theoretic Security*. DOI: 10.1109/SPW.2015.33. 16

T. Durden. 2002. Bypassing PaX ASLR protection. *Phrack*, 11. 10, 17

EEMBC. The embedded microprocessor benchmark consortium: EEMBC benchmark suite. http://www.eembc.org. 206

Ú Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. 2006. XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation*, pp. 75–88. 58, 86, 95

H. Etoh and K. Yoda. June 2000. Protecting from stack-smashing attacks. Technical report, IBM Research Division, Tokyo Research Laboratory. 63

C. Evans. 2013. Exploiting 64-bit Linux like a boss. http://scarybeastsecurity.blogspot.com/2013/02/exploiting-64-bit-linux-like-boss.html. 117

I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. E. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. 2015. Missing the point(er): On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy, (S&P)*, pp. 781–796. DOI: 10.1109/SP.2015.53. 11, 62, 87

I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 901–913. DOI: 10.1145/2810103.2813646. 20, 21, 59, 82, 97, 137, 211

Federal Communications Commission. 2014. Measuring broadband America—2014. http://www.fcc.gov/reports/measuring-broadband-america-2014. 256

C. Fetzer and M. Suesskraut. 2008. SwitchBlade: Enforcing dynamic personalized system call models. In *Proceedings of the 3rd European Conference on Computer Systems*, pp. 273–286. DOI: 10.1145/1357010.1352621. 16

A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina. 2011. SmartDec: Approaching C++ decompilation. In *Working Conference on Reverse Engineering (WCRE)*. 171

B. Ford and R. Cox. 2008. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX ATC*, pp. 293–306. 8, 9

M. Frantzen and M. Shuey. 2001. StackGhost: Hardware facilitated stack protection. In *USENIX Security Symposium*. 139, 141

I. Fratric. 2012. Runtime prevention of return-oriented programming attacks. http://github.com/ivanfratric/ropguard/blob/master/doc/ropguard.pdf. 139, 173, 176

Gaisler Research. LEON3 synthesizable processor. http://www.gaisler.com. 183, 206

A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 465–478. DOI: 10.1145/1543135.1542528. 50

T. Garfinkel, B. Pfaff, and M. Rosenblum. 2004. Ostia: A delegating architecture for secure system call interposition. In *Network and Distributed System Security Symposium (NDSS)*. 241

R. Gawlik and T. Holz. 2014. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Annual Computer Security Applications Conference (ACSAC)*, pp. 396–405. 171, 173, 176, 182

R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. 2016. Enabling client-side crash-resistance to overcome diversification and information hiding. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*. 68

X. Ge, M. Payer, and T. Jaeger. 2017. An evil copy: How the loader betrays you. In *Network and Distributed System Security Symposium (NDSS)*. DOI: 10.14722/ndss.2017.23199 . 15

J. Gionta, W. Enck, and P. Ning. 2015. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp. 325–336. DOI: 10.1145/2699026.2699107. 65

GNU.org. The GNU C library: Environment access. http://www.gnu.org/software/libc/manual/html_node/Environment-Access.html. 220

E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. 2014a. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pp. 575–589. DOI: 10.1109/SP.2014.43. 15, 53, 82, 84, 86, 97, 114, 124, 125, 126, 129, 134, 136, 137, 139, 140, 174, 175, 177, 182, 183, 186, 188, 200, 202, 211

E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. 2014b. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium*. http://dl.acm.org/citation.cfm?id=2671225.2671252. 122, 139, 140, 169, 177, 179, 182, 186, 188, 209

E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. 2016. Undermining information hiding (and what to do about it). In *Proceedings of the 25th USENIX Security Symposium*, pp. 105–119. 11, 68

I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. 1996. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th USENIX Security Symposium (SSYM)*. 16

Google Chromium Project. 2013. Undefined behavior sanitizer. http://www.chromium.org/developers/testing/undefinedbehaviorsanitizer. 7

B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. 2017. ASLR on the line: Practical cache attacks on the MMU. In *Annual Network and Distributed System Security Symposium (NDSS)*. 67

Y. Guillot and A. Gazet. 2010. Automatic binary deobfuscation. *J. Comput. Virol.* 6(3): pp. 261–276. DOI: 10.1007/s11416-009-0126-4. 160

I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. 2015. ShrinkWrap: VTable protection without loose ends. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 341–350. DOI: 10.1145/2818000.2818025. 136

I. Haller, Y. Jeon, H. Peng, M. Payer, H. Bos, C. Giuffrida, and E. van der Kouwe. 2016. TypeSanitizer: Practical type confusion detection. In *ACM Conference on Computer and Communication Security (CCS)*. DOI: 10.1145/2976749.2978405. 7

N. Hasabnis, A. Misra, and R. Sekar. 2012. Light-weight bounds checking. In *IEEE/ACM Symposium on Code Generation and Optimization*. DOI: 10.1145/2259016.2259034. 84

Hex-Rays. 2017. IDA Pro. http://www.hex-rays.com/index.shtml. 128

M. Hicks. 2014. What is memory safety? http://www.pl-enthusiast.net/2014/07/21/memory-safety/. 4

E. Hiroaki and Y. Kunikazu. 2001. ProPolice: Improved stack-smashing attack detection. *IPSJ SIG Notes*, pp. 181–188. 11

J. Hiser, A. Nguyen, M. Co, M. Hall, and J. W. Davidson. 2012. ILR: Where'd my gadgets go? In *33rd IEEE Symposium on Security and Privacy (S&P)*, pp. 571–585. DOI: 10.1109/SP.2012.39. 11, 66

J. D. Hiser, D. Williams, A. Filipi, J. W. Davidson, and B. R. Childers. 2006. Evaluating fragment construction policies for SDT systems. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, pp. 122–132. DOI: 10.1145/1134760.1134778. 8, 9

U. Hölzle, C. Chambers, and D. Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming (ECOOP)*, pp. 21–38. 54

U. Hölzle, C. Chambers, and D. Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 32–43. DOI: 10.1145/143103.143114. 54

A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. 2013. Librando: Transparent code randomization for just-in-time compilers. CCS '13, pp. 993–1004. DOI: 10.1145/2508859.2516675. 58, 59

A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. 2013. Profile-guided automated software diversity. In *IEEE/ACM Symposium on Code Generation and Optimization*, pp. 1–11. DOI: 10.1109/CGO.2013.6494997. 85

P. Hosek and C. Cadar. 2013. Safe software updates via multi-version execution. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*, pp. 612–621. DOI: 10.1109/ICSE.2013.6606607. 214, 256, 257

Petr Hosek and Cristian Cadar. 2015. Varan the unbelievable: An efficient n-version execution framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 339–353. DOI: 10.1145/2694344.2694390. 214, 215, 218, 224, 227, 256, 257

H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. 2015. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium*, pp. 177–192. http://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu. 21

R. Hund, C. Willems, and T. Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 191–205. DOI: 10.1109/SP.2013.23. 82, 86, 141

G. Hunt and D. Brubacher. 1999. Detours: Binary interception of win32 functions. In *Usenix Windows NT Symposium*, pp. 135–143. 232

Intel. 2013. *Intel Architecture Instruction Set Extensions Programming Reference*. http://download-software.intel.com/sites/default/files/319433-015.pdf. 108

Intel.2013. Introduction to Intel memory protection extensions. http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions. 93

Intel. 2013. *Intel 64 and IA-32 Architectures Software Developer's Manual—Combined Volumes 1, 2a, 2b, 2c, 3a, 3b, and 3c*. 178

Intel. 2014. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference, A–Z*. 2014. 223, 224

Itanium C++ ABI. http://mentorembedded.github.io/cxx-abi/abi.html. 32

A. Jaleel. 2007. Memory characterization of workloads using instrumentation-driven simulation—a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. *technical report*. http://www.glue.umd.edu/~ajaleel/workload/. 253

D. Jang, Z. Tatlock, and S. Lerner. 2014. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*. 32, 173, 176

jduck. 2010. The latest Adobe exploit and session upgrading. http://bugix-security.blogspot.de/2010/03/adobe-pdf-libtiff-working-exploitcve.html. 182

T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. 2002. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*. 5, 82, 84, 88, 95

N. Joly. 2013. Advanced exploitation of Internet Explorer 10/Windows 8 overflow (Pwn2Own 2013). http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php. 117, 124, 162

M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. 2012. Branch regulation: Low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual*

International Symposium on Computer Architecture (ISCA). http://dl.acm.org/citation.cfm?id=2337159.2337171. DOI: 10.1109/ISCA.2012.6237009. 182, 209

M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh. 2013. Scrap: Architecture for signature-based protection from code reuse attacks. In *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*, pp. 258–269. DOI: 10.1109/HPCA.2013.6522324. 209

C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pp. 339–348. DOI: 10.1109/ACSAC.2006.9. 11, 85

V. Kiriansky, D. Bruening, and S. P. Amarasinghe. 2002. Secure execution via program shepherding. In *Proceedings 11th USENIX Security Symposium*, pp. 191–206. 8, 9

K. Koning, H. Bos, and C. Giuffrida. 2016. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 431–442. DOI: 10.1109/DSN.2016.46. 211, 214, 217

T. Kornau. 2010. Return oriented programming for the ARM architecture. Ph.D. thesis, Master's thesis, Ruhr-Universitat Bochum. 233

V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. 2014a. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 147–163. 9, 10, 59, 62, 105, 106, 107, 173, 178, 179

V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. 2014b. Code-Pointer Integrity website. http://dslab.epfl.ch/proj/cpi/. 179

V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song. 2015. Poster: Getting the point (er): On the feasibility of attacks on code-pointer integrity. In *36th IEEE Symposium on Security and Privacy (S&P)*. 87

P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. 2014. SoK: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pp. 276–291. DOI: 10.1109/SP.2014.25. 11, 62, 66, 250, 252

C. Lattner and V. Adve. 2005. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *ACM Conference on Programming Language Design and Implementation*, pp. 129–142. DOI: 10.1145/1064978.1065027. 91, 108

C. Lattner, A. Lenharth, and V. Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM Conference on Programming Language Design and Implementation*, pp. 278–289. DOI: 10.1145/1273442.1250766. 91, 108

B. Lee, C. Song, T. Kim, and W. Lee. 2015. Type casting verification: Stopping an emerging attack vector. In *USENIX Security 15*, pp. 81–96. http://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lee. 7

D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. 2010. Respec: Efficient online multiprocessor replayvia speculation and external determinism. *ACM SIGARCH Computer Architecture News*, 38(1):77–90. DOI: 10.1145/1736020.1736031. 230

J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz. 2016. Subversive-C: Abusing and protecting dynamic message dispatch. In *USENIX Annual Technical Conference (ATC)*, pp. 209–221. 70, 140

E. Levy. 1996. Smashing the stack for fun and profit. *Phrack*, 7. 61

J. Li, Z. Wang, T. K. Bletsch, D. Srinivasan, M. C. Grace, and X. Jiang. 2011. Comprehensive and efficient protection of kernel control data. *IEEE Transactions on Information Forensics and Security*, 6(4):1404–1417. DOI: 10.1109/TIFS.2011.2159712. 82, 86

C. Liebchen, M. Negro, P. Larsen, L. Davi, A.-R. Sadeghi, S. Crane, M. Qunaibit, M. Franz, and M. Conti. 2015. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM Conference on Computer and Communications Security (CCS)*. DOI: 10.1145/2810103.2813671. 182, 183, 205

Linux Man-Pages Project. 2017a. tc-netem(8)—Linux manual page. 256

Linux Man-Pages Project. 2017b. shmop(2)—Linux manual page. 247

Linux Programmer's Manual. 2017a. vdso(7)—Linux manual page. 223

Linux Programmer's Manual. 2017b. getauxval(3)—Linux manual page. 224

Linux Programmer's Manual. 2017c. signal(7)—Linux manual page. 225

T. Liu, C. Curtsinger, and E. Berger. 2011. DTHREADS: Efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP)*, pp. 327–336. DOI: 10.1145/2043556.2043587. 230

LLVM. The LLVM compiler infrastructure. http://llvm.org/. 102

K. Lu, X. Zhou, T. Bergan, and X. Wang. 2014. Efficient deterministic multithreading without global barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 287–300. DOI: 10.1145/2555243.2555252. 230

K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. 2015. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *ACM Conference on Computer and Communications Security (CCS)*, pp. 280–291. DOI: 10.1145/2810103.2813694. 68

J. Maebe, M. Ronsse, and K. D. Bosschere. 2003. Instrumenting JVMs at the machine code level. In *3rd PA3CT symposium*, volume 19, pp. 105–107. 222

G. Maisuradze, M. Backes, and C. Rossow. 2003. What cannot be read, cannot be leveraged? Revisiting assumptions of JIT-ROP defenses. In *USENIX Security Symposium*. 67

M. Marschalek. 2014. Dig deeper into the IE vulnerability (cve-2014-1776) exploit. http://www.cyphort.com/dig-deeper-ie-vulnerability-cve-2014-1776-exploit/. 182

A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. 2014. Cryptographically enforced control flow integrity. http://arxiv.org/abs/1408.1451. 86

A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. 2015. CCFI: Cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, pp. 941–951. DOI: 10.1145/2810103.2813676. 72, 76, 77

M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. 2013. System V application binary interface: AMD64 architecture processor supplement. http://x86-64.org/documentation/abi.pdf. 150

M. Maurer and D. Brumley. 2012. Tachyon: Tandem execution for efficient live patch testing. In *USENIX Security Symposium*, pp. 617–630. 214, 256, 257

S. McCamant and G. Morrisett. 2006. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*. 41, 59, 68, 86

H. Meer. 2010. Memory corruption attacks: The (almost) complete history. In *Proceedings of Blackhat USA*. 62

T. Merrifield and J. Eriksson. 2013. Conversion: Multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pp. 127–139. DOI: 10.1145/2465351.2465365. 230

Microsoft Corp. November 2014. Enhanced mitigation experience toolkit (EMET) 5.1. http://technet.microsoft.com/en-us/security/jj653751. 173, 176

Microsoft Developer Network. 2017. Argument passing and naming conventions. http://msdn.microsoft.com/en-us/library/984x0h58.aspx. 149, 151, 154

V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. 2015. Opaque control-flow integrity. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*. http://www.internetsociety.org/doc/opaque-control-flow-integrity. 173, 177, 182

J. R. Moser. 2006. Virtual machines and memory protections. http://lwn.net/Articles/210272/. 238

G. Murphy. 2012. Position independent executables—adoption recommendations for packages. http://people.redhat.com/˜gmurphy/files/pie.odt. 238

S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *International Symposium on Computer Architecture*, pp. 189–200. DOI: 10.1145/2366231.2337181. 109

S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. 2015. Everything you want to know about pointer-based checking. In *First Summit on Advances in Programming Languages (SNAPL)*. DOI: 10.4230/LIPIcs.SNAPL.2015.190. 5

S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM Sigplan Notices*, volume 44, pp. 245–258. DOI: 10.1145/1542476.1542504. 4, 5, 36, 82, 84, 88, 91, 97, 99, 101, 102, 110, 112, 211

S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. 2010. CETS: Compiler enforced temporal safety for C. In *ACM Sigplan Notices*, volume 45, pp. 31–40. DOI: 10.1145/1806651.1806657. 6, 83, 84, 88, 89, 91, 98, 108, 173, 178, 211

G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526. DOI: 10.1145/1065887.1065892. 5, 82, 84, 88, 95

Nergal. December 2001. The advanced return-into-lib(c) exploits (PaX case study). *Phrack*, 58 (4): 54. http://www.phrack.org/archives/58/p58_0x04_Advanced%20return-into-lib(c)%20exploits%20(PaX%20case%20study)_by_nergal.txt. 81, 82, 185, 203

B. Niu. 2015. Practical control-flow integrity. Ph.D. thesis, Lehigh University. 26, 37, 39, 56, 60

B. Niu and G. Tan. 2013. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 199–210. DOI: 10.1145/2508859.2516649. 39, 82, 86, 173, 175

B. Niu and G. Tan. 2014a. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. DOI: 10.1145/2594291.2594295. 26, 27, 40, 44, 58, 82, 110, 114, 182

B. Niu and G. Tan. 2014b. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communication Security (CCS)*, pp. 1317–1328. DOI: 10.1145/2660267.2660281. 9, 26, 34

B. Niu and G. Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 914–926. DOI: 10.1145/2810103.2813644. 14, 30, 59

A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. 2016. Poking holes in information hiding. In *25th USENIX Security Symposium*, pp. 121–138. 11, 68, 94

M. Olszewski, J. Ansel, and S. Amarasinghe. 2009. Kendo: Efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108. DOI: 10.1145/1508244.1508256. 230

K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. 2010. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, pp. 49–58. DOI: 10.1145/1920261.1920269. 173, 177, 210

V. Pappas, M. Polychronakis, and A. D. Keromytis. 2013. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, pp. 447–462. 117, 118, 119, 127, 173, 176, 182, 209

A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. Marx: Uncovering class hiearchies in C++ programs. In *Annual Network and Distributed System Security Symposium (NDSS)*. 67

PaX Team. 2004a. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt, 2004a. 82, 85, 211

PaX Team. 2004b PaX non-executable pp. design & implementation. http://pax.grsecurity.net/docs/noexec.txt, 2004b. 8, 211

M. Payer. 2012. Safe loading and efficient runtime confinement: A foundation for secure execution. Ph.D. thesis, ETH Zurich. http://nebelwelt.net/publications/12PhD. DOI: 10.1109/SP.2012.11. 8

M. Payer, A. Barresi, and T. R. Gross. 2015. Fine-grained control-flow integrity through binary hardening. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. DOI: 10.1007/978-3-319-20550-2_8. 10, 14, 58, 173, 174

M. Payer and T. R. Gross. 2011. Fine-grained user-space security through virtualization. In *Proceedings of the 7th International Conference on Virtual Execution Environments (VEE)*. DOI: 10.1145/1952682.1952703. 8, 9

A. Pelletier. 2012. Advanced exploitation of Internet Explorer heap overflow (Pwn2Own 2012 exploit). VUPEN Vulnerability Research Team (VRT) blog. http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php. 124, 131

J. Pewny and T. Holz. 2013. Control-flow restrictor: Compiler-based CFI for iOS. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 309–318. DOI: 10.1145/2523649.2523674. 58

Phoronix. Phoronix test suite. http://www.phoronix-test-suite.com/. 114

A. Prakash, X. Hu, and H. Yin. 2015. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Symposium on Network and Distributed System Security (NDSS)*. 58, 160, 170, 171, 173, 176, 182

N. Provos. 2003. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium SSYM)*, volume 12, pp. 18–18. http://dl.acm.org/citation.cfm?id=1251353.1251371. 16, 241

H. P. Reiser, J. Domaschka, F. J. Hauck, R. Kapitza, and W. Schröder-Preikschat. 2006. Consistent replication of multithreaded distributed objects. In *IEEE Symposium on Reliable Distributed Systems*, pp. 257–266. DOI: 10.1109/SRDS.2006.14. 230

R. Roemer, E. Buchanan, H. Shacham, and S. Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34. DOI: 10.1145/2133375.2133377. 20, 117, 181, 185

R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi. 2017. Address oblivious code reuse: On the effectiveness of leakage resilient diversity. In *Annual Network and Distributed System Security Symposium (NDSS)*. 69

J. M. Rushby. 1981. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 12–21. DOI: 10.1145/800216.806586. 214

M. Russinovich, D. A. Solomon, and A. Ionescu. 2012. *Windows Internals, Part 1*. Microsoft Press, 6th edition. ISBN 978-0-7356-4873-9. 155, 175

SafeStack. Clang documentation: Safestack. http://clang.llvm.org/docs/SafeStack.html. 102

B. Salamat. 2009. Multi-variant execution: Run-time defense against malicious code injection attacks. Ph.D. thesis, University of California at Irvine. 218

B. Salamat, T. Jackson, A. Gal, and M. Franz. 2009. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, pp. 33–46. DOI: 10.1145/1519065.1519071. 211, 213, 214, 217, 227, 256, 257

J. Salwan. 2011. ROPGadget. http://shell-storm.org/project/ROPgadget/. 136

F. Schuster. July 2015. *Securing Application Software in Modern Adversarial Settings*. Ph.D. thesis, Katholieke Universiteit Leuven. 140

F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy (S&P)*, pp. 745–762. DOI: 10.1109/SP.2015.51. 15, 20, 67, 70, 97, 140, 182, 183, 184, 185, 186, 200, 204, 211

F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. 2014. Evaluating the effectiveness of current anti-ROP defenses. In *Research in Attacks, Intrusions, and Defenses*, volume 8688 of *Lecture Notes in Computer Science*. DOI: 10.1007/978-3-319-11379-1_5. 139, 140, 177, 182, 186, 188, 209

E. J. Schwartz, T. Avgerinos, and D. Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)*. DOI: 10.1109/SP.2010.26. 86

E. J. Schwartz, T. Avgerinos, and D. Brumley. 2011. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security (SEC)*, pp. 25–25. 136

C. Segulja and T. S. Abdelrahman. 2014. What is the cost of weak determinism? In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 99–112. DOI: 10.1145/2628071.2628099. 254

J. Seibert, H. Okhravi, and E. Söderström. 2014. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pp. 54–65. DOI: 10.1145/2660267.2660309. 141, 182

K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pp. 309–318. 82, 84, 173, 178

F. J. Serna. 2012. CVE-2012-0769, the case of the perfect info leak. http://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf. 63, 117

H. Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, pp. 552–561. DOI: 10.1145/1315245.1315313. 62, 172, 184, 185, 186, 200, 233

H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pp. 298–307. DOI: 10.1145/1030083.1030124. 62

N. Shavit and D. Touitou. 1995. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 204–213. 28

K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy (S&P)*, pp. 574–588. DOI: 10.1109/SP.2013.45. 10, 20, 49, 63, 82, 86, 117, 141, 177, 182, 184, 186, 203

K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. 2016. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *37th IEEE Symposium on Security and Privacy (S&P)*, pp. 954–968. DOI: 10.1109/SP.2016.61. 70

Solar Designer. 1997a. "return-to-libc" attack. Bugtraq. 203

Solar Designer.1997b. lpr LIBC RETURN exploit. http://insecure.org/sploits/linux.libc.return.lpr.sploit.html. 203

C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. 2015. Exploiting and protecting dynamic code generation. In *Network and Distributed System Security Symposium (NDSS)*. 49, 58, 60

A. Sotirov. 2007. Heap feng shui in JavaScript. *In Proceedings of Black Hat Europe*. 132

E. H. Spafford. January 1989. The internet worm program: An analysis. *SIGCOMM Comput. Commun. Rev.*, 19 (1): 17–57. ISSN 0146-4833. DOI: 10.1145/66093.66095. 61

SPARC. SPARC V8 processor. http://www.sparc.org. 206

R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. 2009. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security (EUROSEC)*, pp. 1–8. DOI: 10.1145/1519144.1519145. 63, 117

D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin. 2016. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *IEEE/ACM Design Automation Conference (DAC)*, pp. 83.2:1–6. DOI: 10.1145/2897937.2898098. 209

L. Szekeres, M. Payer, T. Wei, and D. Song. 2013. SoK: Eternal war in memory. In *Proceedings International Symposium on Security and Privacy (S&P)*. DOI: 10.1109/SP.2013.13. 2, 61, 82, 85, 211

L. Szekeres, M. Payer, L. Wei, D. Song, and R. Sekar. 2014. Eternal war in memory. *IEEE Security and Privacy Magazine*. DOI: 10.1109/MSP.2013.47. 2

A. Tang, S. Sethumadhavan, and S. Stolfo. 2015. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM Conference on Computer and Communications Security (CCS)*, pp. 256–267. DOI: 10.1145/2810103.2813685. 70

C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*. http://dl.acm.org/citation.cfm?id=2671225.2671285. 58, 86, 173, 175, 182, 204, 208, 211, 233

M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. 2011. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, pp. 121–141. DOI: 10.1007/978-3-642-23644-0_7. 117, 140, 183, 184, 185, 200, 204

A. van de Ven. August 2004. New security enhancements in Red Hat Enterprise Linux v.3, update 3. http://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf. 9, 82

V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. 2015. PathArmor: Practical ROP protection using context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 927–940. DOI: 10.1145/2810103.2813673. 14, 137

V. van der Veen, N. D. Sharma, L. Cavallaro, and H. Bos. 2012. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, pp. 86–106. DOI: 10.1007/978-3-642-33338-5_5. 61

S. Volckaert. 2015. Advanced Techniques for multi-variant execution. Ph.D. thesis, Ghent University. 226, 231

S. Volckaert, B. Coppens, and B. De Sutter. 2015. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Trans. on Dependable and Secure Computing*, 13 (4): 437–450. DOI: 10.1109/TDSC.2015.2411254. 211, 250

S. Volckaert, B. Coppens, B. De Sutter, K. De Bosschere, P. Larsen, and M. Franz. 2017. Taming parallelism in a multi-variant execution environment. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pp. 270–285. DOI: 10.1145/3064176.3064178. 230, 232

S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz. 2016. Secure and efficient application monitoring and replication. In *USENIX Annual Technical Conference (ATC)*, pp. 167–179. 214, 215, 247

S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere. 2013. GHUMVEE: Efficient, effective, and flexible replication. In *5th International Symposium on Foundations and Practice of Security (FPS)*, pp. 261–277. 214, 217, 232

R. Wahbe, S. Lucco, T. Anderson, and S. Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pp. 203–216. DOI: 10.1145/168619.168635. 8, 9, 41, 68, 249

Z. Wang and X. Jiang. 2010. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pp. 380–395. DOI: 10.1109/SP.2010.30. 58, 208

R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 157–168. DOI: 10.1145/2382196.2382216. 11, 173, 177

R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. 2010. Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium*, pp. 29–46. 16

T. Wei, T. Wang, L. Duan, and J. Luo. 2011. INSeRT: Protect dynamic code generation against spraying. In *International Conference on Information Science and Technology (ICIST)*, pp. 323–328. DOI: 10.1109/ICIST.2011.5765261. 59

J. Werner, G. Baltas, R. Dallara, N. Otternes, K. Snow, F. Monrose, and M. Polychronakis. 2016. No-execute-after-read: Preventing code disclosure in commodity software. In *11th ACM Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 35–46. DOI: 10.1145/2897845.2897891. 70

J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. 2011. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 41–50. DOI: 10.1145/2076732.2076739. 109, 239

R. Wojtczuk. 1998. Defeating Solar Designer's non-executable stack patch. http://insecure.org/sploits/non-executable.stack.problems.html. 20, 81, 82, 203

C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. 2002. Linux security modules: General security support for the Linux kernel. In *Proceedings 11th USENIX Security Symposium*. 16

R. Wu, P. Chen, B. Mao, and L. Xie. 2012. RIM: A method to defend from JIT spraying attack. In *7th International Conference on Availability, Reliability, and Security (ARES)*, pp. 143–148. DOI: 10.1109/ARES.2012.11. 59

Y. Xia, Y. Liu, H. Chen, and B. Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, pp. 1–12. DOI: 10.1109/DSN.2012.6263958. 173, 176

F. Yao, J. Chen, and G. Venkataramani. 2013. JOP-alarm: Detecting jump-oriented programming-based anomalies in applications. In *IEEE 31st International Conference on Computer Design (ICCD)*, pp. 467–470. DOI: 10.1109/ICCD.2013.6657084. 209

B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy (S&P)*, pp. 79–93. DOI: 10.1109/SP.2009.25. 8, 39, 86

B. Zeng, G. Tan, and Ú. Erlingsson. 2013. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Security Symposium*, pp. 369–382. 58, 86

B. Zeng, G. Tan, and G. Morrisett. 2011. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 29–40. DOI: 10.1145/2046707.2046713. 58, 59, 86

C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. 2015. VTint: Defending virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)*. DOI: 10.14722/ndss.2015.23099 . 160, 173, 176, 182

C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. 2013. Practical control flow integrity and randomization for binary executables. In *34th IEEE Symposium on Security and Privacy (S&P)*, pp. 559–573. DOI: 10.1109/SP.2013.44. 38, 82, 86, 97, 110, 114, 117, 118, 119, 127, 136, 169, 173, 174, 182, 208, 209

M. Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, pp. 337–352. http://dl.acm.org/citation.cfm?id=2534766.2534796. 38, 82, 86, 97, 110, 114, 117, 118, 119, 127, 136, 173, 174, 182, 208, 209

H. W. Zhou, X. Wu, W. C. Shi, J. H. Yuan, and B. Liang. 2014. HDROP: Detecting ROP attacks using performance monitoring counters. In *Information Security Practice and Experience*, pp. 172–186. Springer International Publishing. DOI: 10.1007/978-3-319-06320-1_14. 173, 177

X. Zhou, K. Lu, X. Wang, and . Li. 2012. Exploiting parallelism in deterministic shared memory multiprocessing. *Journal of Parallel and Distributed Computing*, 72(5):716–727. DOI: 10.1016/j.jpdc.2012.02.008. 230

Contributor Biographies

Editors

Per Larsen is trying his hand as an entrepreneur and co-founded an information security startup—Immunant, Inc.—specializing in exploit mitigation. Previously, he worked for four years as a postdoctoral scholar at the University of California, Irvine. He graduated with a Ph.D. from the Technical University of Denmark in 2011.

Per co-organized the 2015 Dagstuhl Seminar upon which this book is based and has served as program committee member for several academic conferences including USENIX Security, USENIX WOOT, ICDCS, and AsiaCCS. In 2015, he was recognized as a DARPA Riser.

Ahmad Sadeghi is a full professor of Computer Science at TU Darmstadt, Germany, and the Director of the Intel Collaborative Research Institute for Secure Computing (ICRI-SC) at TU Darmstadt. He holds a Ph.D. in computer science from the University of Saarland in Saarbrücken, Germany. Prior to academia, he worked in research and development for telecommunications enterprises, amongst others Ericsson Telecommunications. He is editor-in-chief of *IEEE Security and Privacy Magazine,* and on the editorial board of ACM Books. He served five years on the editorial board of the *ACM Transactions on Information and System Security* (TISSEC), and was guest editor of the *IEEE Transactions on Computer-Aided Design* (*Special Issue on Hardware Security and Trust*).

Authors

Orlando Arias is pursuing his Ph.D. in computer engineering from the University of Central Florida under the advisement of Dr. Yier Jin. His research interests include secure computer architectures, network security, IP core design, and integration and cryptosystems.

Elias Athanasopoulos is an assistant professor with the Computer Science Department at the University of Cyprus. Before joining the University of Cyprus, he was an assistant professor with Vrije Universiteit Amsterdam. He holds a BSc in physics from the University of Athens and a Ph.D. in computer science from the University of Crete. Elias is a Microsoft Research Ph.D. Scholar. He has interned with Microsoft Research in Cambridge and worked as a research assistant with FORTH in Greece from 2005 to 2011. Elias is also a Marie Curie fellow. Before joining the faculty of Vrije Universiteit Amsterdam, he was a postdoctoral research scientist with Columbia University and a collaborating researcher with FORTH.

Herbert Bos is a professor of systems and network security at Vrije Universiteit Amsterdam, where he heads the VUSec research group. He obtained his Ph.D. from Cambridge University Computer Laboratory (UK). Coming from a systems background, he drifted into security a few years ago and never left.

George Candea heads the Dependable Systems Lab at EPFL, where he conducts research on both the fundamentals and the practice of achieving reliability and security in complex software systems. His main focus is on real-world large-scale systems—millions of lines of code written by hundreds of programmers—because going from a small program to a large system introduces fundamental challenges that cannot be addressed with the techniques that work at small scale. George is also Chairman of Cyberhaven, a cybersecurity company he co-founded with his former students to defend sensitive data against advanced attacks, social engineering, and malicious insiders. In the past, George was CTO and later Chief Scientist of Aster Data Systems (now Teradata Aster). Before that, he held positions at Oracle, Microsoft Research, and IBM Research. George is a recipient of the first Eurosys Jochen Liedtke Young Researcher Award (2014), an ERC StG award (2011), and the MIT TR35 Young Innovators award (2005). He received his Ph.D. (2005) in computer science from Stanford and his B.S. (1997) and M.Eng. (1998) in electrical engineering and computer science from MIT.

Bart Coppens is a postdoctoral researcher at Ghent University in the Computer Systems Lab. He received his Ph.D. in computer science engineering from the Faculty of Engineering and Architecture at Ghent University in 2013. His research focuses on protecting software against different forms of attacks using compiler-based techniques and run-time techniques.

Stephen Crane started seriously diving into security during his undergrad at Cal Poly Pomona, competing in CCDC. From there he worked on research somewhere in the intersection of systems security and compilers at UC Irvine. After transforming into Dr. Crane, Stephen founded Immunant with fellow UCI researchers, where he tries to get exploit mitigation tools into the hands of developers.

Lucas Davi is an assistant professor of computer science at University of Duisburg-Essen, Germany, and associated researcher at the Intel Collaborative Research

Institute for Secure Computing (ICRI-SC) at TU Darmstadt, Germany. He received his Ph.D. in computer science from TU Darmstadt. His research focus includes system security, software security, and trusted computing. His Ph.D. thesis on code-reuse attacks and defenses has been awarded with the ACM SIGSAC Dissertation Award 2016.

Bjorn De Sutter is a professor at Ghent University in the Computer Systems Lab. He obtained his MSc. and Ph.D. degrees in computer science from Ghent University's Faculty of Engineering in 1997 and 2002. His research focuses on the use of compiler techniques and run-time techniques to aid programmers with non-functional aspects of their software, such as performance, code size, reliability, and software protection. He has published over 80 peer-reviewed papers on these topics.

Michael Franz is the director of the Secure Systems and Software Laboratory at the University of California, Irvine (UCI). He is a full professor of computer science in UCI's Donald Bren School of Information and Computer Sciences and a full professor of electrical engineering and computer science (by courtesy) in UCI's Henry Samueli School of Engineering. Prof. Franz was an early pioneer in the areas of mobile code and dynamic compilation. He created an early just-in-time compilation system, contributed to the theory and practice of continuous compilation and optimization, and co-invented the trace compilation technology that eventually became the JavaScript engine in Mozilla's Firefox browser. Franz received a Dr. sc. techn. degree in computer science and a Dipl. Informatik-Ing. ETH degree, both from the Swiss Federal Institute of Technology, ETH Zurich.

Enes Göktaş is a Ph.D. Student in the systems and network security group at the Vrije Universiteit Amsterdam. His research focus is on evaluating and developing mitigations against memory corruption vulnerabilities. His previous work includes evaluation and proposal of Control-Flow Integrity based mitigations. His interests lies in the area of software security, as well as binary analysis and instrumentation.

Thorsten Holz is a professor in the Faculty of Electrical Engineering and Information Technology at Ruhr-University Bochum, Germany. His research interests include systems-oriented aspects of secure systems, with a specific focus on applied computer security. Currently, his work concentrates on bots/botnets, automated analysis of malicious software, and studying the latest attack vectors. He received the Dipl.-Inform. degree in computer science from RWTH Aachen, Germany (2005), and a Ph.D. degree from University of Mannheim (2009). Prior to joining Ruhr-University Bochum in April 2010, he was a postdoctoral researcher in the Automation Systems Group at the Technical University of Vienna, Austria. In

2011, Thorsten received the Heinz Maier-Leibnitz Prize from the German Research Foundation (DFG).

Andrei Homescu finished his doctoral studies at UC Irvine in 2015, after which he co-founded Immunant with three of the present co-authors. Andrei has published widely in the areas of systems security, language runtimes, and exploit mitigation. He also led the development of selfrando, a production-ready, open-source randomization engine, and is the lead author on several US patents and patent applications.

Yier Jin received his Ph.D. in electrical engineering from Yale University in 2012. He is an assistant professor in the Department of Electrical and Computer Engineering, University of Central Florida, USA. His research interests include hardware security, IoT security, and formal methods. He is a member of IEEE and ACM.

Volodymyr Kuznetsov is a security researcher who focuses on practical applications of program analysis and other formal methods in systems security, with particular interest in protecting sensitive data in applications and systems with security vulnerabilities. Volodymyr's research was released as open source projects that have become widely used in research community and industry (http://s2e.epfl.ch/, http://clang.llvm.org/docs/SafeStack.html, and others) and were recognized with an open source award. Volodymyr obtained his Ph.D. in Computer Science at EPFL in 2016 and now leads a cyber security company that brings state-of-the-art research into the world of enterprise cyber security.

Ben Niu earned his Ph.D. degree in computer science from Lehigh University, USA, in 2016, advised by professor Gang Tan. He is currently a security software engineer at Microsoft Corporation. His research interests are system security and parallel programming.

Hamed Okhravi is a senior staff member at the Cyber Analytics and Decision Systems Group of MIT Lincoln Laboratory, where he leads programs and conducts research in the area of systems security. His research interests include cyber security, science of security, security evaluation, and operating systems. He is the recipient of the 2014 MIT Lincoln Laboratory Early Career Technical Achievement Award and 2015 Team Award for his work on cyber moving target research. He is also the recipient of an honorable mention (runner-up) at the 2015 NSA's 3rd Annual Best Scientific Cybersecurity Paper Competition. Currently, his research is focused on analyzing and developing system security defenses.

He has served as a program chair for the ACM CCS Moving Target Defense (MTD) workshop and program committee member for a number of academic conferences and workshops including ACM CCS, NDSS, RAID, AsiaCCS, ACNS, and IEEE SecDev.

Dr. Okhravi earned his MS and Ph.D. in electrical and computer engineering from University of Illinois at Urbana-Champaign in 2006 and 2010, respectively.

Mathias Payer is a security researcher and assistant professor in computer science at Purdue University, leading the HexHive group. His research focuses on protecting applications even in the presence of vulnerabilities, with a focus on memory corruption. He is interested in system security, binary exploitation, user-space software-based fault isolation, binary translation/recompilation, and (application) virtualization. All implementation prototypes from his group are open source. In 2014, he founded the b01lers Purdue CTF team. Before joining Purdue in 2014, he spent two years as a postdoc in Dawn Song's BitBlaze group at UC Berkeley. He graduated from ETH Zurich with a Dr. sc. ETH in 2012.

Georgios Portokalidis is an assistant professor in the Department of Computer Science at Stevens Institute of Technology. He obtained his Ph.D. from Vrije Universiteit in Amsterdam in February 2010, and also spent a couple of years as a postdoc at Columbia University in New York. His research interests center mainly around the area of systems and security, including software and network security, authentication, privacy, and software resiliency. His recent work has revolved around code-reuse attacks, efficient information-flow tracking, and security applications using the Internet of Things. During his Ph.D. he worked on the Argos emulator, a platform for hosting high-interaction honeypots that can automatically detect zero-day control-flow hijacking attacks, and Paranoid Android, a record-replay system for the Android OS.

Felix Schuster has been a researcher at the Microsoft Research Cambridge (UK) lab since 2015. Before joining Microsoft Research, he obtained a Ph.D. from Ruhr-Universität Bochum. Felix is broadly interested in applied systems and software security and is part of the lab's Constructive Security Group. In the past, he worked on topics like code-reuse attacks and defenses (e.g., COOP) and automated binary code analysis. Currently, Felix's research focusses on the design of practical solutions for the trusted cloud like the VC3 system or the Coco blockchain framework.

R. Sekar is a Professor of Computer Science and the Director of the Secure Systems Laboratory and the Center for Cyber Security at Stony Brook University. He received his Bachelor's degree in Electrical Engineering from IIT, Madras (India) in 1986, and his Ph.D. in Computer Science from Stony Brook in 1991. He then served as a Research Scientist at Bellcore until 1996, and then as faculty at Iowa State University. Sekar's research interests are focused on software exploit detection and mitigation, malware and untrusted code defense, and security policies and their enforcement.

Dawn Song is a professor in the Department of Electrical Engineering and Computer Science at UC Berkeley. Her research interest lies in deep learning and security. She has studied diverse security and privacy issues in computer systems and networks, including areas ranging from software security, networking security, database security, distributed systems security, and applied cryptography, to the intersection of machine learning and security. She is the recipient of various awards including the MacArthur Fellowship, the Guggenheim Fellowship, the NSF CAREER Award, the Alfred P. Sloan Research Fellowship, the MIT Technology Review TR-35 Award, the George Tallman Ladd Research Award, the Okawa Foundation Research Award, the Li Ka Shing Foundation Women in Science Distinguished Lecture Series Award, the Faculty Research Award from IBM, Google and other major tech companies, and Best Paper Awards from top conferences. She obtained her Ph.D. from UC Berkeley. Prior to joining UC Berkeley as a faculty, she was an Assistant Professor at Carnegie Mellon University from 2002–2007.

Dean Sullivan is pursuing his Ph.D. in computer engineering from the University of Central Florida under the advisement of Dr. Yier Jin. His research interests include system security and computer architecture.

László Szekeres is a software security researcher at Google. He works on developing techniques for protecting against security bugs, primarily in C/C++ code. His research is focused on finding and hardening against vulnerabilities using automated test generation, program analysis, compiler techniques, and machine learning. He obtained his Ph.D. in Computer Science from Stony Brook University in 2017. During his studies he spent a year as a visiting researcher at UC Berkeley. In 2010, he was awarded the Fulbright Foreign Student Scholarship. Before returning to academia for his doctorate degree, he led a security research team at a spin-off company of the Budapest University of Technology and Economics.

Gang Tan received his Ph.D. in computer science from Princeton University in 2005. He is an associate professor in the Department of Computer Science and Engineering, Pennsylvania State University, USA. His research interests include software security, programming languages, and formal methods. He is a member of IEEE and ACM.

Stijn Volckaert received his Ph.D. degree from Ghent University's Faculty of Engineering and Architecture. He is currently a postdoctoral scholar in the Department of Computer Science at the University of California, Irvine. His research interests include security, operating systems, and software protection.