Poster: Getting The Point(er): On the Feasibility of Attacks on Code-Pointer Integrity

Volodymyr Kuznetsov*, László Szekeres†, Mathias Payer‡, George Candea*, and Dawn Song§
*École Polytechnique Fédérale de Lausanne (EPFL), †Stony Brook Universtiy, ‡Purdue University, §UC Berkeley

***Abstract*—Control-flow hijack attacks remain a major security problem, despite many years of research aimed at mitigating them. Code-Pointer Integrity (CPI) [2] is the first protection mechanism that systematically prevents all such attacks while keeping performance overhead low. In the upcoming S&P'15 paper, Evans et al. [1] claim that CPI can be bypassed on x86-64 and ARM architectures. This poster is a clarifying response to the claims in [1] that CPI as a whole has a security weakness.**

The CPI property by itself is secure as shown by a formal correctness proof [2]. Bugs or weaknesses in specific CPI implementations can lead to security weaknesses. We discuss different implementation alternatives, analyze their security guarantees and performance implications, and demonstrate that the attack presented in [1] is only effective against the simplest proof-of-concept implementation of CPI. The presented attack cannot subvert the other implementation alternatives, e.g., the ones using hardware-enforced segmentation or software fault isolation.

I. Code-Pointer Integrity

Code-Pointer Integrity (CPI) [2] is a protection mechanism that prevents all control-flow hijack attacks that are caused by memory corruption errors with low performance overhead. CPI instruments C/C++ programs at compile time, enforcing precise memory safety for all direct and indirect pointers to code in a program, which ensures the above security guarantee. CPI comes with a proof of correctness outlined in [2]. Code-Pointer Separation (CPS) is a simplified version of CPI that provides practical protection against most control-flow hijack attacks by ensuring the integrity of direct pointers to code only. The performance overhead of our CPI implementation on SPEC2006 benchmarks is 8.4% on average, while the performance overhead of CPS is 1.9% on average.

CPI achieves low performance overhead by limiting memory safety enforcement to sensitive pointers only (i.e., direct or indirect pointers to code). The key idea is to split program memory into two isolated regions: the safe region stores all sensitive pointers, and the regular region stores everything else. CPI uses static analysis to identify program instructions that may access the safe region, and instruments them with memory safety checks. CPI employs instruction-level isolation in order to prevent all other instructions from accessing the safe region, even if hijacked by the attacker. In order to avoid changing the memory layout of the regular region, CPI reserves the locations normally occupied by sensitive pointers in the regular region and, in the safe region, it maintains a map from the addresses of these reserved locations to pointer values and corresponding metadata required for memory safety checks.

An implementation of CPI or CPS consists of (i) a static analysis pass that splits all memory accesses in a program into those that may access sensitive pointers and those that cannot, (ii) an instrumentation pass that instruments all accesses that might access sensitive pointers to use the safe region and inserts runtime checks that enforce memory safety of these accesses, and (iii) an instruction-level isolation mechanism that prevents all memory accesses that are not instrumented with memory safety checks from ever accessing the safe region, even if a memory access is compromised by an attacker. This poster focuses on the third step, as do Evans et al. [1].

We implemented and presented in [2] multiple mechanisms that efficiently enforce instruction-level isolation using either hardware-enforced segmentation, software fault isolation, or randomization and information hiding. The security guarantees and performance implications of these mechanisms are summarized in Table I, 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.

The attack in [1] focuses on one of these mechanisms, namely the one based on information hiding, and it is only effective against its simplest proof-of-concept implementation.

A. Hardware-Enforced Segmentation Based Implementation

On architectures that support hardware-enforced segmentation, 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 prove to be indeed valuable.

This implementation of CPI is precise and imposes zero performance overhead on instructions that do not access sensitive pointers. It is not vulnerable to the attack in [1].

B. 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

	Security		Overhead	
	CPI	CPS	CPI	CPS
Hardware segmentation	precise		8.4%	1.9%
Software fault isolation	precise		13.8%	7.0%
Information hiding				
- hashtable	16.6	20.7	9.7%	2.2%
- lookup table	15	17	8.9%	2.0%
- linear table	5	7	8.4%	1.9%

TABLE I. Security guarantees (either precise or number of entropy bits) and performance overhead (average on SPEC2006) of various implementations of CPI/CPS

software fault isolation (SFI). In our implementation, we align the safe region in memory so that enforcing a pointer to not 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 [2].

Our SFI-based implementation of CPI is precise, and SFI increases the overhead by less then 5% relative to hardware-enforced segmentation. It is not vulnerable to the attack in [1].

C. 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 of 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 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 hashtable, a two-level lookup table, or a linear table to organize the safe region [2]. 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 (which is consistent with the experimental evaluation in [2]), which amounts to $1\,\text{GByte} \times 8\% \,/\, 8\,\text{bytes} \approx 2^{23.4}$ sensitive pointers in total.

Hashtable 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 hashtable would occupy $2^{23.4}/0.25 \times 32 = 2^{30.4}$ bytes of memory. Randomizing the hashtable 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 a 8×2^{22} bytes region of the address space. Assuming sensitive pointers are uniformly distributed across the 1GB of continuous program memory, CPI will allocate $1\,\text{GByte}/(8 \times 2^{22}) \times 32 \times 2^{22} = 2^{32}\,\text{bytes}$ for the subtables. Randomizing the subtables 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.

Linear 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 linear table in this implementation can be guessed while causing only 16 crashes on average.

Code-Pointer Separation, unlike full CPI, does not require any metadata and has fewer sensitive pointers (by $8.5\times$ on average [2]). This increases the number of expected crashes by $17\times$ for the hashtable-based implementation, and by $4\times$ for the other two implementations.

The information-hiding-based implementation of CPI that uses a hashtable to organize the safe region provides probabilistic security guarantees with $2^{16.6}$ bits of entropy (or $2^{20.7}$ for CPS). The attack in [1] would cause $\approx 51,000$ crashes (or $\approx 867,000$ for CPS) on average before the location of the safe region is detected and the attack succeeds. We believe that 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.

II. The attack on CPI's simplest implementation

Evans et al. [1] present an attack against a CPI implementation based on information hiding that uses a linear table to organize the safe memory region, one of the five implementations of CPI described above. The attack is based on a remote side channel information leak vulnerability in the nginx web server that enables an attacker to remotely read arbitrary locations in nginx memory with high confidence. The attack first uses this arbitrary memory read capability in order to probe program memory and locate the safe region, exploiting a limitation of the proof-of-concept implementation of CPI to further reduce the number of crashes below 16. Once the address of the safe memory region is known, the attack employs another memory write vulnerability in order to overwrite the value of a sensitive pointer directly in the safe region, which is not detected by CPI.

As discussed in the previous section, this attack is only effective against the simplest of the CPI implementations outlined in Table I. We believe that the security of CPI as a whole is not weakened by the existence of this attack, and discouraging wide use of CPI would be a disservice to the community.

References

[1] Isaac Evans, Sam Fingeret, Julián González, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *IEEE Symp. on Security and Privacy*, 2015.

[2] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Symp. on Operating Systems Design and Implementation*, 2014.