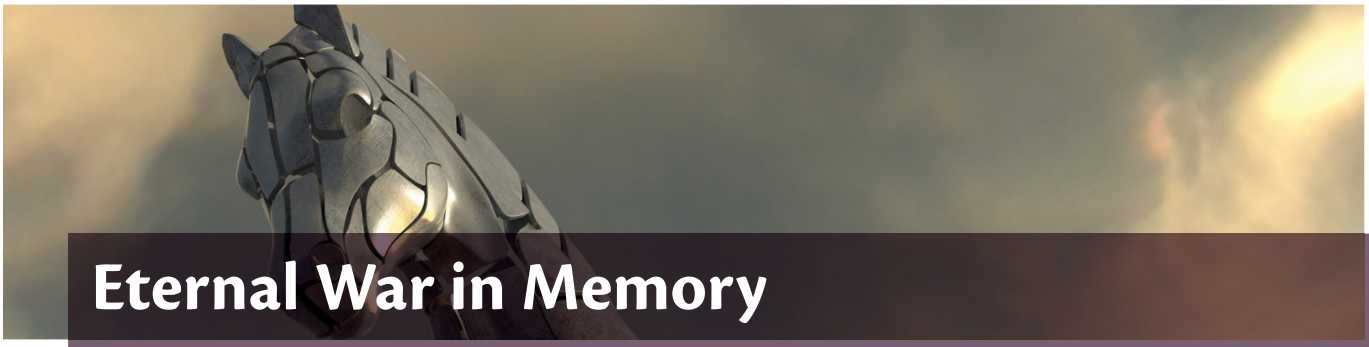

Eternal War in Memory

László Szekeres | Stony Brook University
Mathias Payer | University of California, Berkeley
Lenx Tao Wei | FireEye
R. Sekar | Stony Brook University

Software written in low-level languages like C or C++ is prone to memory corruption bugs that allow attackers to access machines, extract information, and install malware. Real-world exploits show that all widely deployed protections can be defeated.

Most software we use every day, such as browsers, office suites, PDF viewers, and operating systems, are written in low-level languages such as C or C++. These languages give programmers explicit and fine-grained control over memory, making writing very efficient code possible. However, they can introduce memory-related bugs that let attackers alter a program's behavior and take full control over a program or system.

In this article, we give an overview of the "eternal war in memory," providing a model of memory errors and identifying the different general security policies that can mitigate the attacks. Because of the limited space, we mention only a few concrete mechanisms implementing and enforcing these policies. Our paper, "SoK: Eternal War in Memory," provides more complete coverage of the tools as well as more technical details.[1]

Background

Clearly, the best way to get rid of memory corruption bugs is to fix them. Unfortunately, finding these bugs is hard, and fixing them requires a lot of manual work. Another option is to avoid low-level languages and rewrite all vulnerable applications in type-safe languages. This is unrealistic as well, owing to the billions of lines of C and C++ code in use today. Therefore, we focus on automatic solutions that prevent exploitation without modifying source code. There's a tradeoff between fixing a bug and using an automatic mechanism to prevent exploitation: when a security mechanism detects an attack, it terminates the program, whereas when performing a manual fix, a programmer can handle the error and recover from it.

The war in memory is fought by researchers developing defense mechanisms and attackers finding new ways around these protections. This 30-year-old arms race between offense and defense continues today. Each year, hackers demonstrate how systems can be compromised in contests such as Pwn2Own or Pwnium. For example, in 2013, hackers successfully exploited memory bugs in Adobe Reader, Adobe Flash, Oracle Java, and Windows 8 in the latest releases of Chrome, Internet Explorer, and Firefox.

Researchers have developed defense mechanisms protecting applications from different forms of

1540-7993/14/$31.00 © 2014 IEEE Copublished by the IEEE Computer and Reliability Societies

attacks—some widely deployed in commodity systems and compilers. Stack cookies, exception handler validation, data execution prevention (DEP or W⊕X), and address space layout randomization (ASLR) make successful exploitation of memory corruption bugs much harder. Unfortunately, several attack vectors are still effective under all currently deployed protection settings, including return-oriented programming (ROP),[2] information leaks, and just-in-time (JIT) code reuse.[3] New defense mechanisms have been proposed but aren't widely deployed, often owing to high performance overhead, incompatibility with legacy code, and nonrobust protection.

Attacks

To analyze and compare protection mechanisms, we first need to understand the attack process. Here, we set up a model of all memory corruption exploits, breaking them into simple steps. Later, we discuss policies obstructing the steps in these attacks as well as the protection mechanisms enforcing each policy.

In Figure 1, each beige rectangular node represents a step of an exploit that leads to a successful attack—code corruption, control-flow hijacking, data-only, or information leak—represented by red nodes. Each diamond represents a decision between paths to the goal. Control-flow hijacking is often the primary attack goal, but memory corruption can be used to carry out other types of attacks, such as information leaks and data-only attacks.

Memory Corruption

The first two steps of an exploit cover the initial memory error (see Figure 1). The first step makes a pointer invalid, and the second dereferences the pointer, thereby triggering the error. We consider a pointer invalid when it goes out of the bounds of its target object or when the object is deallocated. A pointer that points to a deleted object is called a *dangling pointer*. Dereferencing an out-of-bounds pointer causes a *spatial error*, whereas dereferencing a dangling pointer causes a *temporal error*.

Attackers can execute these first two steps using typical programming bugs in low-level languages, such as

- buffer overflow/indexing bug: an attacker controls the index into an array but the bounds check is missing or faulty, leading to a spatial error (this bug might result from an integer overflow or faulty type conversion); and
- use-after-free bug: a pointer is used after its pointed object has been freed, causing a temporal error.

When read or written, out-of-bounds or dangling pointers can cause corruption or leakage of internal data (step 3). When an out-of-bounds pointer is dereferenced to read a value into a register, the value is corrupted. Consider the following jump table in which the function pointer defining the next function call is read from an array without a bounds check:

```
func_ptr jump_table[3] =
   {fn0,fn1,fn2};
   jmp_table[user_input]();
// call *(jmp_table+user_input).
```

By providing an invalid index as `user_input`, attackers can alter the target pointer to a location under their control and thus read an unintended value, which diverts the control flow. Besides corrupting data, reading memory through an attacker-specified pointer can leak information if that data is included in the output. A classic example of this attack is a *format string bug*. By specifying the format string, attackers can create unexpected pointers that the `printf` function will use:

```
printf (user_input);
   // input "%3$x" creates an invalid
   // pointer and prints the third
   // integer on the stack.
```

When attackers use an out-of-bounds pointer to write, they can overwrite anything in memory, including regular data, other pointers, and executable code. Attackers often exploit buffer overflows and indexing bugs to overwrite internal data such as a return address or an object's virtual table (vtable) pointer. Vtable pointer corruption is an example of the backward loop in step 3 of Figure 1. Suppose an array pointer is made out of bounds (step 1) so that when it's dereferenced to write (step 2), it points to a vtable pointer (step 3) that attackers forge to point to a location that they control (step 4). When the corrupted vtable pointer is later dereferenced to look up a called virtual function (step 5), a bogus function pointer is loaded (step 3 of the control-flow hijack attack). With a single memory error, attackers can trigger a cascade of follow-up memory errors by corrupting other pointers. Attackers can also exploit write dereferences to leak information. For example, they can leak arbitrary memory contents in this line of code by corrupting the `err_msg` pointer, similarly to format string bugs:

```
printf("%s\n",err_msg);
```

Temporal errors—caused by a dereferenced dangling pointer (step 2)—can be exploited similarly to spatial errors. In temporal errors, the deallocated object's memory space is, at least partially, reused by

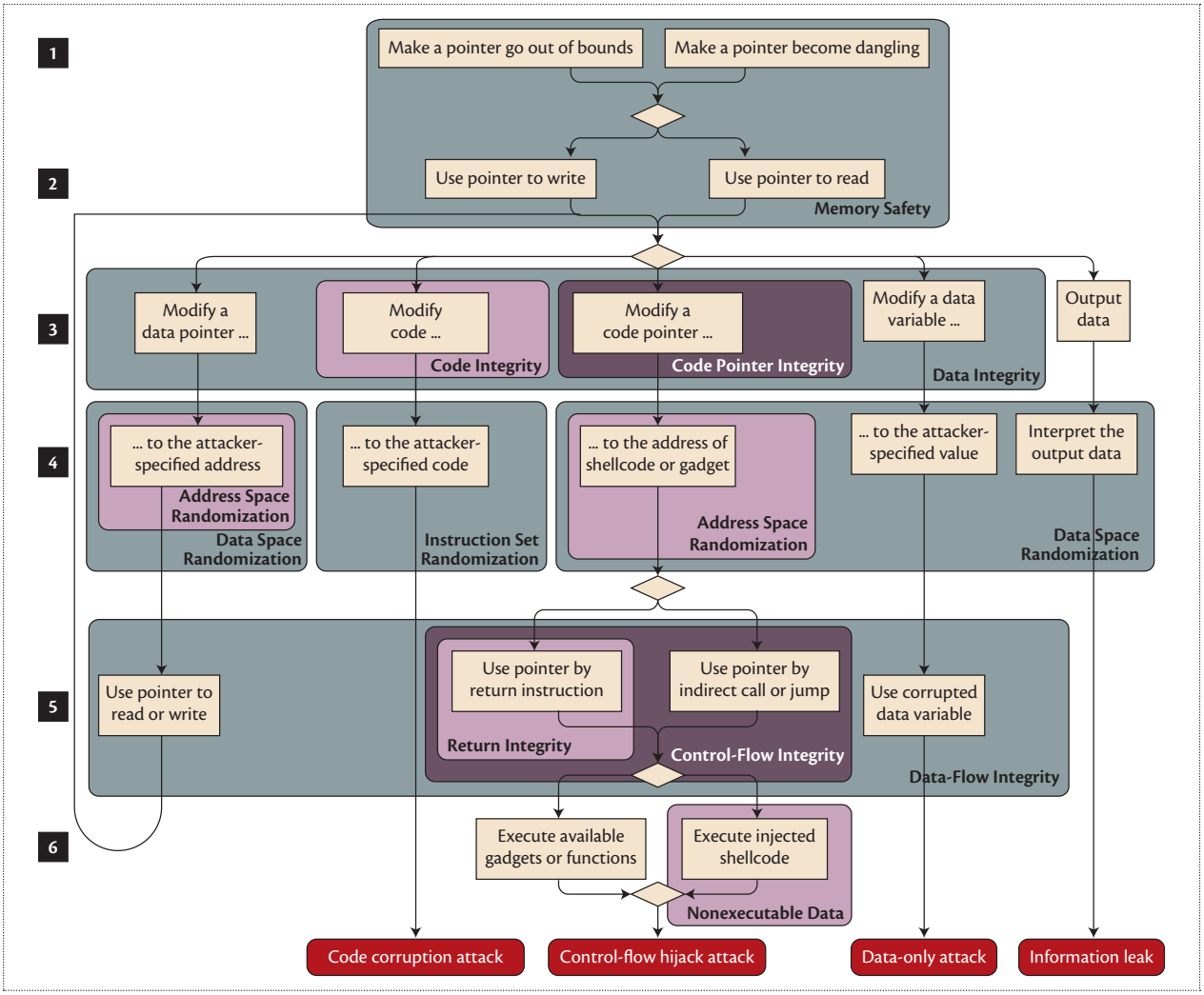

Figure 1. Memory corruption attacks and policies mitigating them. Each beige rectangular node represents a step of an exploit that leads to a successful attack, represented by red nodes. Each diamond represents a decision between alternative paths to the goal. The policy that mitigates a given set of attack steps is represented by a dark purple area surrounding the beige boxes. Policies surrounded by a light purple box are enforced by current protections.

another object. A type mismatch between the old and new object also lets attackers access otherwise inaccessible memory locations.

Let's first consider reading through a dangling pointer with the old object's type but pointing to some newly allocated buffer holding untrusted user input. When a virtual function of the old object is called, that object's vtable pointer loads from a location that's now under an attacker's control; the attacker-provided data will be interpreted as the old object's vtable pointer. This is comparable to exploiting a spatial write error to overwrite the vtable pointer inside an object, but in this case, only a read dereference is exploited. In addition, sensitive information inside the newly allocated object can be leaked when read through the dangling pointer of the old object's type.

Attackers can exploit writing through a dangling pointer in a similar fashion, especially when there's a type mismatch. A pointer pointing to the stack can also become dangling, for instance, if a function returns a pointer to a local variable. When the local variable is later written through the dangling pointer, it might point to a saved return address in a different, newly allocated stack frame. *Double-free* is a special case of a use-after-free vulnerability, when `free()` is called on a dangling pointer. This can lead to corruption of the memory manager's internal data structures, which attackers can exploit to corrupt more pointers.

Attackers can use any combination of the first two steps in the model to both corrupt internal data and leak sensitive information. Furthermore, they can trigger additional memory errors with other corrupted pointers. In general, memory corruption lets attackers read and modify a program's internal state in unintended ways. The errors we described are a violation of the Memory Safety policy (see Figure 1).

C and C++ are inherently memory-unsafe because they don't prohibit these errors. Bounds checking and memory management are the programmers' responsibility. According to the C and C++ standards, writing an array beyond its bounds, dereferencing a null pointer, or reading an uninitialized variable doesn't necessarily have to raise an exception or error, but the result is undefined.

Code Corruption

The most obvious way to modify a program's execution is to use a bug to overwrite the program code in memory. The Code Integrity policy ensures that program code can't be modified and is typically enforced by setting all code pages to read-only after the program is loaded into memory. All modern processors support this feature.

Control-Flow Hijacking

Most often, attackers exploit memory corruption to control program execution by diverting its control flow. Control-flow hijack attacks use memory errors to corrupt a code pointer (step 3). The Code Pointer Integrity policy prevents code pointer corruption. It can't be enforced as easily as Code Integrity because most code pointers, such as return addresses, must be stored in writable memory, for instance, on the stack or heap. Furthermore, even if all code pointers are read-only, they might be corrupted when being read into a register through a corrupted pointer. The example we gave for reading through an out-of-bounds pointer demonstrates this. Currently, no published protection technique enforces this policy.

Suppose attackers carry out step 3 and modify a saved return address. To hijack the control flow, they also need to know where to divert it—that is, the target's location (step 4). The Address Space Randomization (ASR) policy randomizes the potential targets' locations. An attack fails if the attacker can't guess the correct address.

Suppose attackers successfully corrupt a code pointer, such as a function pointer, in the first four steps. Step 5 is to load the pointer into the program counter register. In most architectures, the program counter or instruction pointer is updated only indirectly with the execution of an indirect control-flow transfer instruction, such as an indirect function call, indirect jump, or function return. Diverting the execution from the control flow defined by the source code violates the Control-Flow Integrity (CFI) policy.

The final step of a control-flow hijack exploit is the execution of malicious code (step 6). Classic attacks inject shellcode into memory and divert execution to this piece of code. The Nonexecutable Data policy prevents such code injection. Combining Nonexecutable Data and Code Integrity results in the W⊕X policy: a page can be either writable or executable, but not both. Most modern CPUs support nonexecutable page permissions, and enforcing W⊕X is cheap and practical.

To bypass the Nonexecutable Data policy, attackers reuse existing code in memory. The reused code can be existing functions, as in the case of a return-to-libc attack, or *gadgets*—small instruction sequences found anywhere in the code that can be chained together to carry out useful operations. This approach is called *return-oriented programming* because functions or gadgets are often chained by return instruction. Attackers can use other indirect jump instructions for this purpose, and sometimes no chaining is necessary—for instance, calling `system()` with an arbitrary command. At this point, the attack is successful: attacker-specified code is executed. We don't cover higher-level policies, such as sandboxing, system call interposition, and file permissions, which only confine attackers' power or capabilities. Although limiting the damage that attackers can cause after compromising a program is important, we focus on preventing the compromise.

Data-Only

Control-flow hijacking isn't the only way to carry out an attack. In general, attackers want to maliciously modify program logic to gain more control, gain privileges, or leak information. They can achieve these goals without modifying data explicitly related to the control flow. For instance, consider the modification of the `isAdmin` variable via a buffer overflow:

```
bool isAdmin = false;
...
if (isAdmin) // do privileged operations.
```

These are called *non-control-data attacks* because neither code nor code pointers (control data) are corrupted.[4] The target of corruption is any security-critical data in memory, for example, configuration data, user IDs, or cryptographic keys.

The steps for this attack are comparable to control-data attacks. Here, the goal is to corrupt a security-critical variable in step 3. Because any data could be

security critical, the integrity of all variables must be protected to prevent an attack. We call this policy Data Integrity, generalizing Code Integrity and Code Pointer Integrity. Data Integrity approaches prevent the corruption of data in memory by enforcing an approximation of the Memory Safety policy.

As with code pointers, attackers must know the corrupted data's new value. Introducing entropy into the representation of all data using the Data Space Randomization (DSR) policy can prevent attackers from knowing the correct values. DSR techniques generalize and extend ASR by introducing entropy in not only memory addresses (pointers) but also data variables.

Similar to code pointer corruption, data-only attacks succeed only when the corrupted variable is used. In our running example, the `if (isAdmin)` statement must successfully execute without detecting the corruption. As a generalization of the CFI policy, using any corrupted data (not only corrupted pointers) violates the Data-Flow Integrity (DFI) policy.

Information Leak

Any type of memory error can be exploited to leak memory contents that are otherwise excluded from the output. Information leaks are most often used to circumvent probabilistic defenses. In a threat model in which attackers have complete access to the memory due to memory corruption, the only policy beyond Memory Safety that might mitigate information leakage is full DSR, which encrypts data in memory.

Control-Flow Protections

Here, we cover techniques enforcing the policies that mitigate the steps contributing to control-flow hijack attacks—Return Integrity, Code Integrity and Nonexecutable Data, ASR, and CFI. Stack cookies, W⊕X, and ASLR are currently the only widely deployed protection mechanisms. In Figure 1, the policies enforced by these mechanisms are depicted with a light purple box. No widely used technique enforces CFI, but the ongoing research is promising.

Return Integrity

Stack cookies, also called *canaries*, were the first defense against "stack smashing" attacks wherein attackers exploit a buffer overflow on the stack to overwrite a return address. The defense aims to enforce the integrity of return addresses and other control data, such as saved base pointers, by placing a secret value, or cookie, between the return address and local variables. The return address can only be overwritten by a contiguous buffer overflow if the cookie is overwritten as well. By checking whether the cookie value has changed before returning, the attack can be detected.

Return Integrity is weaker than CFI because it doesn't protect indirect calls and jumps (step 5). Furthermore, stack cookie protection is limited even for returns: return addresses are vulnerable to direct overwrites—for example, exploiting an indexing error—and cookies are vulnerable to information leaks. However, stack cookies are popular and widely deployed because the performance overhead is negligible (typically less than 1 percent), and no compatibility issues are introduced.

Code Integrity and Nonexecutable Data

W⊕X—the combination of Code Integrity and Nonexecutable Data—protects against code corruption and code injection but doesn't protect against code reuse like ROP in step 6. This protection is widely used and has negligible overhead due to hardware support. Unfortunately, W⊕X isn't compatible with self-modifying code or JIT compilation. Every major browser and office application includes a JIT compiler for JavaScript, ActionScript, or VBScript. The dynamic code's integrity can't be enforced because there's a time window during which the generated code is on a writable page. But more important, the code is generated from an attacker-provided source (for example, in JavaScript), which can be exploited to produce useful gadgets for code reuse.[5]

Address Space Randomization

ASLR is the most prominent memory ASR technique. It randomly arranges the position of each code and data memory area. If the payload's address in the virtual memory space isn't fixed, attackers can't divert control flow reliably. ASLR is the most comprehensive deployed protection against hijacking attacks. It also mitigates other attack types that involve corrupting more data pointers in memory, because attackers don't know what to change the pointer to (see the backward loop in Figure 1).

Many implementations have serious weaknesses. Often, some memory segments aren't randomized, and attackers can use gadgets from fixed segments. Other times, the introduced entropy (for example, in 32-bit address spaces) isn't effective against brute-force or heap-spray attacks in which the memory is filled with copies of the payload. Several proposed enhancements make randomization more fine grained and increase entropy in both data and code locations by permuting functions, basic blocks, and instructions.

Information leaks—the fundamental attack vector against all probabilistic techniques—can completely undermine ASR.[6] The prevalence of user scripting and JIT compilation makes exploiting information leaks much easier. An attacker-specified script can exploit an information leak to circumvent ASR, use the information to dynamically discover or produce useful code

gadgets, and launch a code-reuse exploit on the fly.[3] ASLR results in less than 10 percent performance overhead on average.

Control-Flow Integrity

CFI enforces control-flow transfers' integrity by checking their targets' validity, as opposed to Return Integrity, which checks only function returns. CFI relies on prior knowledge of all valid targets for each control-flow transfer, including calls, jumps, and returns. For instance, we can use static pointer analysis to establish the valid targets, or points-to sets, of indirect function calls and returns. By assigning different IDs to distinct points-to sets, we mark target locations with the IDs and check the marks before the control transfer instruction. In the first CFI solution, Martin Abadi and his colleagues proposed placing the IDs inside the code to protect them through Code Integrity.[7] The IDs are encoded into instructions that don't affect the code semantics. All indirect calls and returns are instrumented to check whether the target address has the correct ID before jumping there. This mechanism relies on Nonexecutable Data to prevent attackers from forging valid targets by simply placing an ID before an injected shellcode.

This method overapproximates programs' original control-flow graph. First, because of the conservativeness of any pointer analysis, the resulting points-to sets are overapproximations. Second, to have unique IDs, points-to sets that include a common target must be merged. Furthermore, all exported functions in shared libraries must be marked with the same ID because of potential external aliasing.

A weaker but more practical policy restricts indirect control transfers to the union of all their points-to sets. The original CFI implementation and newer, more practical solutions, such as CCFIR[8] and binCFI,[9] use this approach. All indirectly callable functions are marked with the same ID. The advantage of this policy is that it doesn't need pointer analysis; enumerating all functions whose addresses are taken is sufficient.

The main drawback of the original and many other CFI solutions is that they're incompatible with untransformed libraries. If the library functions aren't marked as valid targets, calling them will raise false positives. Performance overhead for CFI implementations is between 5 and 45 percent for most programs.

Generic Protections

Here, we discuss some of the protection mechanisms enforcing policies that provide more comprehensive protection mitigating more than one attack type in our model: Memory Safety, Data Integrity, DSR, and DFI. Currently, none of these techniques is used in practice.

Memory Safety

Memory Safety mitigates memory corruption by preventing both spatial and temporal errors. Type-safe languages enforce this policy by disallowing pointer arithmetic, checking object bounds at array accesses, and using automatic garbage collection instead of manual memory management. To enforce a similar policy for C and C++, which allow pointer arithmetic and manual memory management, the objects' bounds and allocation information must be tracked. This metainformation is associated with either pointers or objects; perfect Memory Safety can be achieved only in the former case.

To enforce spatial safety, some C alternatives such as CCured and Cyclone use *fat pointers*. They extend the pointer representation to a structure that includes the lowest and highest valid values—that is, the pointed object's start and end address—along with the pointer's current value. The primary problem with this approach is that fat pointers break binary compatibility, so protected programs can't use unmodified libraries.

SoftBound addresses this problem by splitting the metadata from the pointer.[10] The pointers are mapped to their bounds information using a hash table or *shadow memory*, a simple linear mapping of the original address space. The code is instrumented to propagate the metadata and check the bounds whenever a pointer is dereferenced. At pointer initialization, the bounds are set to the pointed object's start and end address. Dereference checks ensure that each pointer is inside its bounds. Pointer-based bounds checking stops all spatial errors in step 2 of our exploit model. SoftBound is formally proven to provide this protection.

Maintaining not only bounds but also allocation information with the pointers allows enforcing temporal safety and thus full Memory Safety. Allocation information indicates when referenced objects are still valid. Keeping an extra bit associated with each pointer indicating the object's validity isn't sufficient because all pointers referencing that object must be found and updated when the object is freed. Compiler Enforced Temporal Safety (CETS) extends SoftBound and solves this problem by storing an object's validity in a global dictionary. New objects get a unique ID as the key to the dictionary, and pointers are associated with this ID. A special data structure for the dictionary allows quick and easy object invalidation and fast lookups to check object validity.

CETS is formally proven to enforce temporal safety if spatial safety is enforced. Therefore, combining SoftBound and CETS enforces Memory Safety. These guarantees have a performance price: the SoftBound/CETS instrumentation slows programs by 100 to 300 percent. Alternative techniques mitigate temporal errors' exploitability by ensuring the deallocated objects'

memory space is reused only by objects of the same type (SafeCode and Cling) or by finer-grained heap randomization (DieHard(er)).

Data Integrity

Data Integrity is an approximation of Memory Safety, enforced by associating metainformation with objects instead of pointers. Knowing an object's address isn't enough to determine whether a pointer dereference targets the correct object. Therefore, some object-based techniques (for example, Jones and Kelly's) check pointer arithmetic (step 1) instead of dereferences (step 2) to check bounds. The checks ensure that a pointer referencing an object stays in the object's bounds during pointer arithmetic.

Tools such as Valgrind's Memcheck, Light-weight Bounds Checker (LBC),[11] and Google's AddressSanitizer (ASAN) also track objects, but they don't check pointer arithmetic. They mark the active object's location in a shadow memory space and ensure that dereferenced pointers point to valid objects. By leaving space between objects, they can detect contiguous buffer overflows but not indexing bugs or corruption inside objects.

Even with pointer arithmetic checks, these approaches typically don't detect data corruption inside objects; thus, spatial safety isn't fully enforced. They can't enforce full temporal safety either. They detect accesses to currently deallocated locations, but if another object reuses the location, use-after-free bugs remain undetected. Although these techniques can and do mitigate the exploitation of use-after-free bugs—for example, by delaying the reuse of freed memory regions—provable temporal safety needs a pointer-based approach.

The performance overhead for such tools can often be more than 100 percent. To decrease the overhead and increase the precision of object-based techniques without pointer arithmetic checking, some researchers leverage static analysis. First, many unnecessary checks can be eliminated statically. Second, by using points-to analysis, each pointer dereference can be restricted to access only objects in its own points-to set. Write integrity testing (WIT) calculates distinct points-to sets for every pointer dereferenced for a write and associates an ID with its point-to set.[12] These IDs mark the objects in the shadow memory area and are checked before each indirect write.

A drawback to enforcing different points-to sets is that this approach is incompatible with shared libraries. The established points-to sets depend on the whole program, which means that different programs would need, for instance, different C libraries using different IDs. The only way to remain compatible is to use a single ID (for "marked"), which degenerates to the policy enforced by Memcheck or ASAN. Because WIT doesn't protect reads, data—including function pointers—can be corrupted when read into a register through a corrupt pointer. To compensate for this limitation, WIT statically establishes and checks indirect calls' targets to properly enforce CFI. Because WIT doesn't deal with temporal errors, overwriting a return address via an escaped dangling pointer is possible; however, such bugs are rare in practice. WIT's reported performance overhead is between 10 and 25 percent.

Data Space Randomization

Pointer encryption falls between ASR and DSR. PointGuard encrypts all pointers in memory and decrypts them before they're loaded into registers.[13] This has similar effect as ASLR, as it introduces entropy in addresses, but it does so in the "data space," encrypting the stored address—that is, the pointer value. Pointers are decrypted using XOR with the same key for all pointers. Because it uses only one key, attackers can recover the key if one encrypted pointer is leaked from memory.[6] PointGuard isn't widely adopted because it's neither binary nor source code compatible.

Full DSR overcomes PointGuard's limitations and provides stronger protection.[14] It encrypts every variable, not just pointers, and uses different keys. For a variable v, a key or mask m_v is generated. The code is instrumented to mask and unmask variables, using XOR, when they're stored and loaded from memory. Because different variables can be stored and loaded by the same pointer dereference, variables in equivalent points-to sets must use the same key. Therefore, DSR uses the same pointer analysis as WIT to compute points-to sets. The protection DSR offers is stronger than PointGuard, because encrypting all variables protects against not only control-flow hijacks but also data-only exploits. Using multiple keys makes DSR much more robust against information leaks, yet they remain possible.[6]

As in case of CFI and WIT, establishing the points-to sets depends on the whole program's static knowledge. Because of potential aliasing, we're forced to use only one key or ID to instrument an independent shared library. In this case, the robustness is reduced to the same guarantees that PointGuard offers. Protected binaries are also incompatible with unmodified libraries. Variables encrypted by the transformed module won't be decrypted by the untransformed libraries. DSR's average overhead is 15 to 25 percent.

Data-Flow Integrity

DFI detects data corruption before the data is used by checking memory read targets.[15] DFI restricts reads based on the last instruction that wrote the read location. In program analysis terms, DFI enforces *reaching definition* sets. An instruction's reaching definition set

Table 1. Protection policies and their techniques' performance impact, compatibility issues, and robustness problems.*

	Policy	Technique	Performance overhead (%)	Compatibility	Weakness
Hijack	Return Integrity	Stack cookies	< 5	Good	Direct overwrite and information leaks
	Code Integrity and Nonexecutable Data	Page flags	< 1	Good	Just-in-time compilation
	Address Space Randomization	Address space layout randomization (ASLR)	< 10	Good	Information leaks
	Control-Flow Integrity	Control-Flow Integrity (CFI)	10–45	Libraries	Overapproximation
Generic	Memory Safety	SoftBound and Compiler Enforced Temporal Safety (Softbound+CETS)	100–300	Good	None
	Data Integrity	Write integrity testing (WIT)	10–25	Libraries	Overapproximation, sub objects, use-after-free bugs, and reads
	Data Space Randomization	Data Space Randomization (DSR)	15–25	Libraries	Overapproximation and information leaks
	Data-Flow Integrity	Data-Flow Integrity (DFI)	100–200	Libraries	Overapproximation

**The colors express positive and negative properties: green means favorable, yellow indicates an acceptable shortcoming, and red indicates an issue likely preventing deployment in production environments.*

is the set of instructions that might have last written the value used by the given instruction based on the control-flow graph.

For instance, the DFI policy ensures that the `isAdmin` variable was last written by the write instruction that the source code defines—and not by a rogue attacker-controlled write—and that the return address used by a return was last written by the corresponding call instruction. DFI builds on static points-to analysis to compute the global reaching definition sets. The resulting reaching definition sets are assigned a unique ID. Each written memory location is marked in the shadow memory with the writing instruction ID. For each read, DFI checks that the ID is valid.

Similar to all solutions relying on pointer analysis, independent transformation of shared libraries is an issue for this policy. DFI isn't binary compatible either, because the lack of metadata maintenance in unprotected libraries can cause false alarms. DFI's performance overhead can be 100 to 200 percent.

Dynamic taint analysis is a simplified version of DFI and doesn't require static analysis. A written memory location is marked in the shadow area if the written data is derived from untrusted user input. Upon reading sensitive data, such as a function pointer, the taint analysis instrumentation can check the mark to ensure it doesn't explicitly depend on user data. However, taint analysis often suffers from false positives, and the performance cost can be even higher than DFI without hardware support.

Approach Summary

Table 1 summarizes the approaches we covered, indicating their performance cost, compatibility issues, and weaknesses. The upper half of the table covers protections against control-flow hijack attacks only, and the lower half covers approaches that mitigate memory corruption exploits in general, including the four attacks we identified in our model.

The indicated performance overheads are rough estimates for the worst case. They're based on reported results measured with the SPEC CPU 2000/2006 benchmarks. The judgment of performance overheads is subjective; we categorize overhead as unacceptable only if it doubles the runtime.

The first three approaches in the table show the widely deployed protection mechanisms. They have practically no performance overhead or compatibility issues, but they have significant security weaknesses. Attackers can divert the control flow at an indirect call or jump, unprotected by stack cookies; reuse existing code with W⊕X enforced; and circumvent ASLR by exploiting information leaks. Information leaks are easier to exploit today due to the prevalence of user scripting, which makes the effectiveness of newer, even finer-grained, randomization techniques unclear.

None of the other techniques are perfect regarding robustness, except enforcing complete Memory Safety with pointer-based techniques. Data Integrity solutions, such as object-based techniques, don't provide perfect protection against sub object and use-after-free

corruption. Furthermore, these techniques often check only writes—not reads—to decrease performance overhead. For instance, WIT allows corrupting a value when it's read into a register through an invalid pointer. Tools such as LBC and ASAN are even less precise—as they can be considered single ID versions of WIT—and they have even higher performance costs. Similar to ASLR as a hijack protection, DSR provides the most comprehensive solutions as a generic protection, but both can be circumvented by information leaks.

The robustness of solutions relying on static pointer analysis—that is, CFI, WIT, DSR, and DFI—is bounded by the conservative approximation of points-to sets. However, the bigger problem with these solutions is their issues with shared libraries. Independent transformation of shared libraries is supported only if the overapproximation goes to the extreme, for example, using only one ID/key. Another problem is the incompatibility with unmodified shared libraries that don't maintain the metadata that the transformed code uses and requires. These problems prevent the deployment of CFI and Data Integrity solutions, despite their acceptable overhead.

None of the current solutions solve this 30-year-old problem with low overhead or without compatibility issues. We expect newer, more practical techniques enforcing stronger policies in the future. In other words: the war is not over. ■

References

1. L. Szekeres et al., "SoK: Eternal War in Memory," *Proc. 2013 IEEE Symp. Security and Privacy*, 2013, pp. 48–62.
2. H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," *Proc. 14th ACM Conf. Computer and Communications Security*, 2007, pp. 552–561.
3. K.Z. Snow et al., "Just-in-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," *Proc. 2013 IEEE Symp. Security and Privacy*, 2013, pp. 574–588.
4. S. Chen et al., "Non-Control-Data Attacks Are Realistic Threats," *Proc. 14th Conf. Usenix Security Symp.*, vol. 14, 2005, p. 12.
5. D. Blazakis, "Interpreter Exploitation," *Proc. 4th Usenix Conf. Offensive Technologies*, 2010, pp. 1–9.
6. R. Strackx et al., "Breaking the Memory Secrecy Assumption," *Proc. 2nd European Workshop System Security*, 2009, pp. 1–8.
7. M. Abadi et al., "Control-Flow Integrity," *Proc. 12th ACM Conf. Computer and Communications Security*, 2005, pp. 340–353.
8. C. Zhang et al., "Practical Control Flow Integrity and Randomization for Binary Executables," *Proc. 2013 IEEE Symp. Security and Privacy*, 2013, pp. 559–573.
9. M. Zhang and R. Sekar, "Control Flow Integrity for COTs Binaries," *Proc. 22nd Usenix Security Symp.*, 2013, pp. 337–352.
10. S. Nagarakatte et al., "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," *SIGPLAN Notices*, vol. 44, no. 6, 2009, pp. 245–258.
11. N. Hasabnis, A. Misra, and R. Sekar, "Light-Weight Bounds Checking," *Proc. 10th Int'l Symp. Code Generation and Optimization*, 2012, pp. 135–144.
12. P. Akritidis et al., "Preventing Memory Error Exploits with WIT," *Proc. 2008 IEEE Symp. Security and Privacy*, 2008, pp. 263–277.
13. C. Cowan et al., "PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities," *Proc. 12th Conf. Usenix Security Symp*. vol. 12, 2003, p. 7.
14. S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," *Proc. 14th Conf. Usenix Security Symp.*, vol. 14, 2005, p. 17.
15. M. Castro, M. Costa, and T. Harris, "Securing Software by Enforcing Data-Flow Integrity," *Proc. 7th Symp. Operating Systems Design and Implementation*, 2006, pp. 147–160.

László Szekeres is a PhD candidate in the Secure Systems Laboratory at Stony Brook University. His research interests include automated bug finding and software hardening via program analysis and compiler and language-based techniques. Szekeres received an MS from the Budapest University of Technology and Economics. Contact him at lszekeres@cs.stonybrook.edu.

Mathias Payer is a postdoctoral researcher in the BitBlaze group at University of California, Berkeley. His research interests include system security, binary analysis, user-space software-based fault isolation, binary translation, application virtualization, and compiler-based enforcement of security policies. Payer received a DSc in computer science from ETH Zurich. Contact him at mathias.payer@nebelwelt.net.

Lenx Tao Wei is senior staff research scientist at FireEye. His research interests include mobile security, software analysis and system protection, Web trust and privacy, and programming languages. Wei received a PhD in computer science from Peking University. Contact him at lenx.wei@gmail.com.

R. Sekar is a professor of computer science and director of the Secure Systems Laboratory and the Center for Cyber Security at Stony Brook University. His research interests include software exploit detection and mitigation, malware and untrusted code defense, and security policies and their enforcement. Sekar received a PhD in computer science from Stony Brook University. Contact him at sekar@seclab.cs.stonybrook.edu.