2013 IEEE Symposium on Security and Privacy

# SoK: Eternal War in Memory

László Szekeres[†], Mathias Payer[‡], Tao Wei[*‡], Dawn Song[‡]
[†]*Stony Brook University*
[‡]*University of California, Berkeley*
[*]*Peking University*

***Abstract*—Memory corruption bugs in software written in low-level languages like C or C++ are one of the oldest problems in computer security. The lack of safety in these languages allows attackers to alter the program's behavior or take full control over it by hijacking its control flow. This problem has existed for more than 30 years and a vast number of potential solutions have been proposed, yet memory corruption attacks continue to pose a serious threat. Real world exploits show that all currently deployed protections can be defeated.**

**This paper sheds light on the primary reasons for this by describing attacks that succeed on today's systems. We systematize the current knowledge about various protection techniques by setting up a general model for memory corruption attacks. Using this model we show what policies can stop which attacks. The model identifies weaknesses of currently deployed techniques, as well as other proposed protections enforcing stricter policies.**

**We analyze the reasons why protection mechanisms implementing stricter polices are not deployed. To achieve wide adoption, protection mechanisms must support a multitude of features and must satisfy a host of requirements. Especially important is performance, as experience shows that only solutions whose overhead is in reasonable bounds get deployed.**

**A comparison of different enforceable policies helps designers of new protection mechanisms in finding the balance between effectiveness (security) and efficiency. We identify some open research problems, and provide suggestions on improving the adoption of newer techniques.**

## I. Introduction

Memory corruption bugs are one of the oldest problems in computer security. Applications written in low-level languages like C or C++ are prone to these kinds of bugs. The lack of memory safety (or type safety) in such languages enables attackers to exploit memory bugs by maliciously altering the program's behavior or even taking full control over the control-flow. The most obvious solution would be to avoid these languages and to rewrite vulnerable applications in type-safe languages. Unfortunately, this is unrealistic not only due to the billions of lines of existing C/C++ code, but also due to the low-level features needed for performance critical programs (e.g. operating systems).

The war in memory is fought on one side by offensive research that develops new attacks and malicious attackers, and on the other side by defensive researchers who develop new protections and application programmers who try to write safe programs. The memory war effectively is an arms race between offense and defense. According to the MITRE ranking [1], memory corruption bugs are considered one of the top three most dangerous software errors. Google Chrome, one of the most secure web browsers written in C++, was exploited four times during the Pwn2Own/Pwnium hacking contests in 2012.

In the last 30 years a set of defenses has been developed against memory corruption attacks. Some of them are deployed in commodity systems and compilers, protecting applications from different forms of attacks. Stack cookies [2], exception handler validation [3], Data Execution Prevention [4] and Address Space Layout Randomization [5] make the exploitation of memory corruption bugs much harder, but several attack vectors are still effective under all these currently deployed basic protection settings. Return-Oriented Programming (ROP) [6], [7], [8], [9], [10], [11], information leaks [12], [13] and the prevalent use of user scripting and just-in-time compilation [14] allow attackers to carry out practically any attack despite all protections.

A multitude of defense mechanisms have been proposed to overcome one or more of the possible attack vectors. Yet most of them are not used in practice, due to one or more of the following factors: the *performance* overhead of the approach outweighs the potential protection, the approach is not *compatible* with all currently used features (e.g., in legacy programs), the approach is not *robust* and the offered protection is not complete, or the approach *depends* on changes in the compiler toolchain or in the source-code while the toolchain is not publicly available.

With all the diverse attacks and proposed defenses it is hard to see how effective and how efficient different solutions are and how they compare to each other and what the primary challenges are. The motivation for this paper is to systematize and evaluate previously proposed approaches. The systematization is done by setting up a general model for memory corruption vulnerabilities and exploitation techniques. The defense techniques are classified by the exploits they mitigate and by the particular phase of exploit they try to inhibit. The evaluation is based on robustness, performance and compatibility. Using this evaluation, we also discuss common criteria that need to be fulfilled for successful deployment of a new software defense.

*Corresponding author.


DOI 10.1109/SP.2013.13

Some related work already covers different memory corruption attacks [15], provides historical overview [16] or lists different protection mechanisms [17]. This systematization of knowledge paper extends the related survey papers by developing a new general model for memory corruption attacks and evaluating and comparing proposed defense mechanisms using a new set of criteria that incorporates real-world adoption as well. The paper does not aim to cover or refer every proposed solution, but rather identifies and analyzes the main approaches in a systematical way, sorts out the most promising proposals and points out fundamental problems and unsolved challenges.

With this systematization of knowledge paper we make the following contributions:

- develop a general model of memory corruption attacks and identify different enforceable security policies based on the model;
- clarify what attack vectors are left unprotected by currently used and previously proposed protections by matching their enforced polices with separate phases of different exploits;
- evaluate and compare proposed solutions for performance, compatibility, and robustness;
- discuss why many proposed solutions are not adopted in practice and what the necessary criteria for a new solution are.

The paper is organized as follows. Section II sets up the main model of attacks and classifies protections based on the policies they enforce. Section III discusses currently deployed protections and their main weaknesses. Our evaluation criteria are set up in Section IV, and are used through the analysis of defense approaches covered by the following four sections. Section IX summarizes with a comparative analysis and Section X concludes the paper.

## II. Attacks

To solve the problem of memory error based attacks, we first need to understand the process of carrying out such an exploit. In this section we set up a step-by-step memory exploitation model. We will base our discussion of protection techniques and the policies they enforce on this model. Figure 1 shows the different steps of exploiting a memory error. Each white rectangular node represents a building block towards successful exploitation and the oval nodes on the bottom represent a successful attack. Each rhombus represents a decision between alternative paths towards the goal. While control-flow hijacking is usually the primary goal of attacks, memory corruption can be exploited to carry out other types of attacks as well.

### A. Memory corruption

Since the root cause of all vulnerabilities discussed in this systematization of knowledge paper is memory corruption, every exploit starts by triggering a memory error. The first two steps of an exploit in Figure 1 cover the initial memory error. The first step makes a pointer *invalid*, and the second one dereferences the pointer, thereby triggering the error. A pointer can become invalid by going out of the bounds of its pointed object or when the object gets deallocated. A pointer pointing to a deleted object is called a *dangling* pointer. Dereferencing an out-of-bounds pointer causes a so called *spatial error*, while dereferencing a dangling pointer causes a *temporal error*.

As Step 1, an attacker may force a pointer out of bounds by exploiting various programming bugs. For instance, by triggering an *allocation failure* which is unchecked, the pointer can become a *null pointer* (in kernel-space null-pointer dereferences are exploitable [18]). By excessively incrementing or decrementing an array pointer in a loop without proper bound checking, a *buffer overflow/underflow* will happen. By causing *indexing bugs* where an attacker has control over the index into an array, but the bounds check is missing or incomplete, the pointer might be pointed to any address. Indexing bugs are often caused by integer related errors like an *integer overflow*, truncation or signedness bug, or incorrect pointer casting. Lastly, pointers can be corrupted using the memory errors under discussion. This is represented by the backward loop in Figure 1.

As an alternative, the attacker may make a pointer "dangling" by, for instance, exploiting an incorrect exception handler, which deallocates an object, but does not reinitialize the pointers to it. Temporal memory errors are called *use-after-free* vulnerabilities because the dangling pointer is dereferenced (used) after the memory area it points to has been returned (freed) to the memory management system. Most of the attacks target heap allocated objects, but pointers to a local variable can "escape" as well from the local scope when assigned to a global pointer. Such escaped pointers become dangling when the function returns and the local variable on the stack is deleted.

Next, we show how either an out-of-bounds or a dangling pointer can be exploited to execute any third step in our exploitation model when the invalid pointer is *read* or *written* in Step 2. The third step is either the corruption or the leakage of some internal data.

When a value is *read* from memory into a register by dereferencing a pointer controlled by the attacker, the value can be corrupted. Consider the following jump table where the function pointer defining the next function call is read from an array without bounds checking.

```
func_ptr jump_table[3] = {fn_0, fn_1, fn_2};
jump_table[user_input]();
```

The attacker can make the pointer point to a location under his or her control and divert control-flow. Any other variable read indirectly can be vulnerable.

Besides data corruption, reading memory through an attacker specified pointer leaks information if that data is

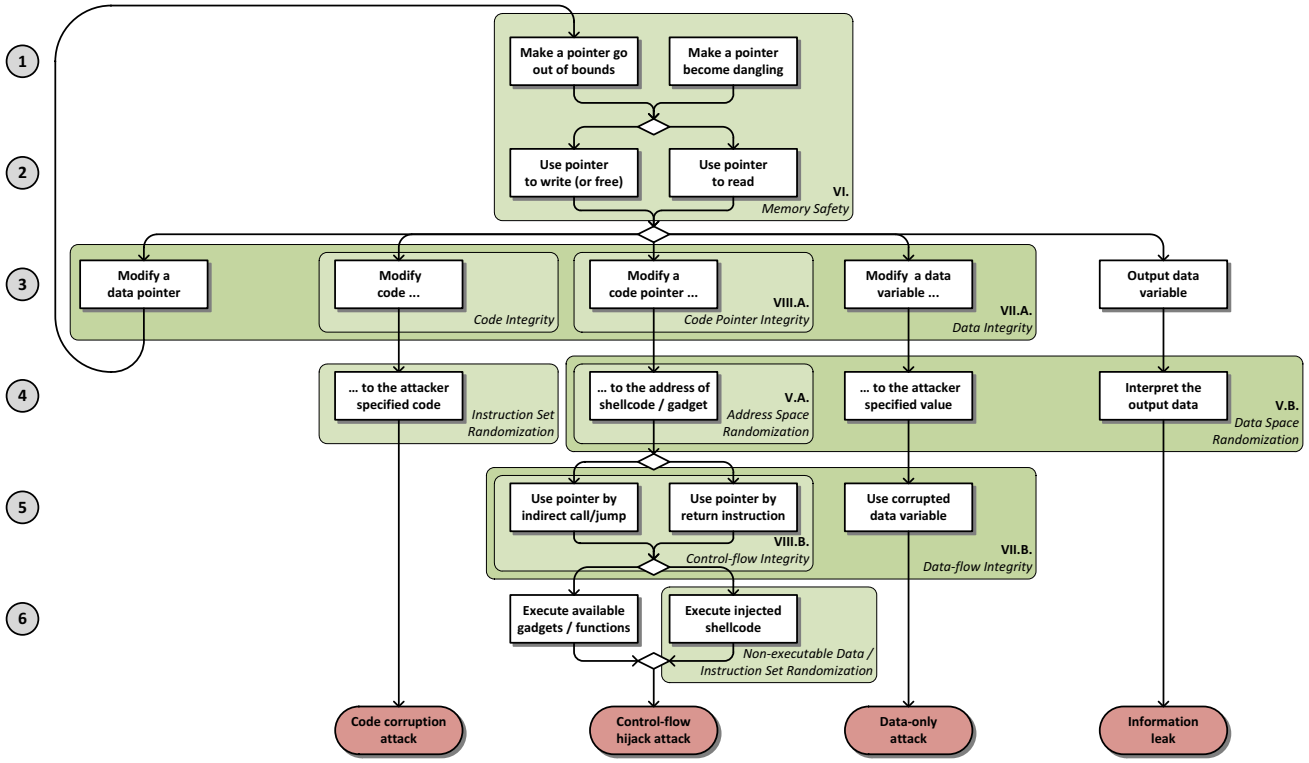


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

included in the output. The classic example of this attack is the `printf` *format string bug*, where the format string is controlled by the attacker. By specifying the format string the attacker creates invalid pointers and reads (and writes) arbitrary memory locations.

```
printf(user_input); // input "%3$x" prints the
                    // 3rd integer on the stack
```

If an attacker controlled pointer is used to *write* the memory, then any variable, including other pointers or even code, can be overwritten. Buffer overflows and indexing bugs can be exploited to overwrite sensitive data such as a return address or virtual table (vtable) pointer. Corrupting the vtable pointer is an example of the backward loop in Figure 1. Suppose a buffer overflow makes an array pointer out of bounds in the first round that is exploited (in Step 3) to corrupt a nearby vtable pointer in memory in the second round. When the corrupted vtable pointer is dereferenced (in Step 2), a bogus virtual function pointer will be used. It is important to see that with one memory error, more and more memory errors can be raised by corrupting other pointers. Calling `free()` with an attacker controlled pointer can also be exploited to carry out arbitrary memory writes [19]. Write dereferences can be exploited to leak information as well.

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

For instance, the attacker is able to leak arbitrary memory contents in the above line of code by corrupting the `err_msg` pointer.

Temporal errors, when a dangling pointer is dereferenced in Step 2, can be exploited similarly to spatial errors. A constraint for exploitable temporal errors is that the memory area of the deallocated object (the old object) is reused by another object (new object). The type mismatch between the old and new object can allow the attacker to access unintended memory.

Let us consider first reading through a dangling pointer with the old object's type but pointing to the new object, which is controlled by the attacker. When a virtual function of the old object is called and the virtual function pointer is looked up, the contents of the new object will be interpreted as the vtable pointer of the old object. This allows the corruption of the vtable pointer, comparable to exploiting a spatial write error, but in this case the dangling pointer is only dereferenced for a read. An additional aspect of this attack is that the new object may contain sensitive information that can be leaked when read through the dangling pointer of the old object's type.

Writing through a dangling pointer is similarly exploitable as an out of bounds pointer by corrupting other pointers or data inside the new object. When the dangling pointer is an escaped pointer to a local variable and points to the stack, it may be exploited to overwrite sensitive data, such as a return address. *Double-free* is a special case of the use-after-free vulnerability where the dangling pointer is used to call `free()` again. In this case, the attacker controlled contents of the new object will be interpreted wrongly as heap metadata, which is also exploitable for arbitrary memory writes [19].

Memory errors in general allow the attacker to read and modify the program's internal state in unintended ways. We showed that *any* combination of the first two steps in our memory exploitation model can be used both to corrupt internal data and to leak sensitive information. Furthermore, more memory errors can be triggered by corrupting other pointers. Programming bugs which make these errors possible, such as buffer overflows and double-frees, are common in C/C++. When developing in such low-level languages, both bounds checking and memory management are fully the programmers responsibility, which is very error prone.

The above described errors are a violation of the *Memory Safety* policy. C and C++ are inherently memory unsafe. According to the C/C++ standards, writing an array beyond its bounds, dereferencing a null-pointer, or reading an uninitialized variable result in *undefined* behavior. Since finding and fixing all the programming bugs is infeasible, we need automatic solutions to enforce Memory Safety in existing programs or stop attacks in their later phases. The policy mitigating a given set of attack steps is represented in the figure by a colored area surrounding the white boxes. The section number which discusses approaches enforcing a given policy is also indicated in the figure, above the policy names. We discuss approaches that try to stop any exploit in the first (two) steps by enforcing Memory Safety in Section VI. In the following subsections we discuss the steps of different exploit paths and identify the policies mitigating the given step. As shown in the figure, some policies include other, weaker policies.

### B. Code corruption attack

The most obvious way to modify the execution of a program is to use one of the abovementioned bugs to overwrite the program code in memory. A *Code Integrity* policy enforces that program code cannot be written. Code Integrity can be achieved if all memory pages containing code are set read-only, which is supported by all modern processors. Unfortunately, Code Integrity does not support self-modifying code or Just-In-Time (JIT) compilation. Today, every major browser includes a JIT compiler for JavaScript or Flash. For these use-cases, Code Integrity cannot be fully enforced because there is a time window during which the generated code is on a writable page.

### C. Control-flow hijack attack

Most often, memory corruption exploits try to take control over the program by diverting its control-flow. If Code Integrity is enforced then this alternative option tries to use a memory error to corrupt a code pointer in Step 3. *Code Pointer Integrity* policies aim to *prevent* the corruption of code pointers. We discuss the potential code pointer targets and limitations of this policy in Section VIII-A.

Suppose the attacker can access and modify a return address due to a buffer overflow. For a successful exploit, the attacker needs to know the correct target value (i.e., the address of the payload) as well. We represent this as a separate fourth step in Figure 1. If the target of the control-flow hijack (the code address to jump to) is not fixed, the attacker cannot specify the target and the attack fails at this step. This property can be achieved by introducing entropy to memory addresses using *Address Space Randomization*. We discuss techniques randomizing the address space in Section V-A.

Suppose a code pointer (e.g., a function pointer) has been successfully corrupted in the first four steps. The fifth step is that the execution needs to load the corrupted pointer into the instruction pointer. The instruction pointer can only be updated indirectly by executing an indirect control-flow transfer instruction, e.g., an indirect function call, indirect jump or function return instruction. Diverting the execution from the control-flow defined by the source code is a violation of the *Control-flow Integrity* (CFI) policy. In Section VIII-B, we cover protections that enforce different CFI policies by *detecting* corruption at indirect control transfers.

The final step of a control-flow hijack exploit is the execution of attacker specified malicious code. Classic attacks injected so-called shellcode into memory, and diverted execution to this piece of code. This kind of exploitation is prevented by the *Non-executable Data* policy which can be enforced using the executable bit for memory pages to make data memory pages, like the stack or the heap, non-executable. A combination of Non-executable Data and Code Integrity results in the W$\oplus$X (Write XOR Execute) [4] policy, stating that a page can be either writable or executable, but not both. Practically all modern CPU support setting non-executable page permissions, so combined with non-writable code, enforcing W$\oplus$X is cheap and practical. However in the case of JIT compilation or self-modifying code, W$\oplus$X cannot be fully enforced. For the sake of completeness, we note that another randomization approach, Instruction Set Randomization (ISR) can also mitigate the execution of injected code or the corruption of existing code by encrypting it. But due to the support for page permissions, the much slower ISR has become less relevant and because of the limited space we will not cover it in more detail in this paper.

To bypass the non-executable data policy, attackers can reuse existing code in memory. The reused code can be an existing function ("return-to-libc" attack) or small instruction sequences (gadgets) found anywhere in the code that can be chained together to carry out useful (malicious) operations. This approach is called Return Oriented Programming (ROP), because the attack chains the execution of functions or gadgets using the ending return instructions. Jump Oriented Programming (JOP) is the generalization of this attack which leverages indirect jumps as well for chaining. There is no policy which can stop the attack at this point, since the execution of valid and already existing code cannot be prevented. Recent research focuses on mitigating techniques against code reuse only. Researchers propose techniques to eliminate useful code chunks (for ROP) from the code by the compiler [20], or by binary rewriting [21]. While these solutions make ROP harder, they do not eliminate all useful gadgets, and they do not prevent re-using complete functions. For these reasons we will not cover these techniques in more detail.

We classify a control-flow hijacking attack successful as soon as attacker-specified code starts to execute. To carry out a meaningful attack, the attacker usually needs to make system calls, and may need high level permissions (e.g., file access) as well. We will not cover higher-level policies which only confine the attacker's access, including permissions, mandatory access control or sandboxing policies enforced by SFI [22], XFI [23] or Native Client [24]. These policies can limit the damage that an untrusted program (or plugin) or an attacker can cause *after* compromising a trusted program. Our focus is preventing the compromise of trusted programs, typically with extensive access (e.g., an ssh/web server).

### D. Data-only attack

Hijacking control-flow is not the only possibility for a successful attack. In general, the attacker's goal is to maliciously modify the program logic to gain more control, to gain privileges, or to leak information. This goal can be achieved without modifying data that is expliclity related to control-flow. Consider, for instance, the modification of the `isAdmin` variable via a buffer overflow after logging into the system with no administrator privileges.

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

These program specific attacks are also called "non-control-data attacks" [25] since neither code nor code pointers (control data) are corrupted. The target of the corruption can be any security critical data in memory, e.g., configuration data, the representation of the user identity, or keys.

The steps of this attack are similar to the previous one except for the target of corruption. Here, the goal is to corrupt some security critical variable in Step 3. Since security critical is a semantic definition, the integrity of all variables has to be protected in order to stop the attack in this step. We call this policy *Data Integrity*, which naturally includes Code Integrity and Code Pointer Integrity. Data Integrity approaches try to *prevent* the corruption of data by enforcing only some approximation of Memory Safety. We cover techniques enforcing such policies under VII-A.

As in the case of code pointers, the attacker needs to know what should replace the corrupted data. Acquisition of this knowledge can be prevented by introducing entropy into the representation of *all* data using *Data Space Randomization*. Data Space Randomization techniques extend and generalize Address Space Randomization, and we cover them in Section V-B.

Similar to code pointer corruption, data-only attacks will succeed as soon as the corrupted variable is used. Using the running example, the `if (isAdmin)` statement has to be successfully executed without *detecting* the corruption. As the generalization of Control-flow Integrity, the use of *any* corrupted data is the violation of *Data-flow Integrity*. We cover enforcing this policy under Section VII-B.

### E. Information leak

We showed that any type of memory error might be exploited to leak memory contents, which would otherwise be excluded from the output. This is typically used to circumvent probabilistic defenses based on randomization and secrets. Real-world exploits bypass ASLR using information leaks [13], [26]. The only policy beyond Memory Safety that might mitigate information leakage is full Data Space Randomization. We will discuss in Section V how effective Data Space Randomization is and how information leakage can be used to bypass other probabilistic techniques which build on secrets.

## III. Currently used protections and real world exploits

The most widely deployed protection mechanisms are stack smashing protection, DEP/W$\oplus$X and ASLR. The Windows platform for instance, also offers some special mechanisms e.g., for protecting heap metadata and exception handlers (SafeSEH and SEHOP).

Stack smashing protection [2] detects buffer overflows of local stack-based buffers, which overwrite the saved return address. By placing a random value (called cookie or canary) between the return address and the local buffers at function entries, the integrity of the cookie can be checked before the return of the function and thus the overflow can be detected. SafeSEH and SEHOP also validate exception handler pointers on the stack before they are used, which makes them, together with stack cookies, a type of Control-flow Integrity solution. These techniques provide the weakest protection: they place checks only before a small subset of indirect

jumps, focusing checking the integrity of only some specific code pointers, namely saved return addresses and exception handler pointers on the stack. Furthermore, the checks can be bypassed. Cookies, for instance, can detect a buffer overflow attack but not a direct overwrite (e.g., exploiting an indexing error).

DEP/W⊕X can protect against code injection attacks, but does not protect against code reuse attacks like ROP. ROP exploits can be generated automatically [27], and large code bases like the C library usually provide enough gadgets for Turing-completeness [10], [11]. ASLR provides the most comprehensive protection as the most widely deployed Address Space Randomization technique. It can randomize the locations of various memory segments, including data and code, so even if the attacker wants to reuse a gadget its location will be random. While some ASLR implementations have specific weaknesses (e.g., code regions are left in predictable locations, de-randomization attacks are possible due to the low entropy), the fundamental attack against it is information leakage [12].

As described in the attack model of the previous section, any memory corruption error can be converted into an information leak vulnerability, which can be used to obtain current code addresses. The leaked addresses are needed to construct the final exploit payload. When attacking remote targets (i.e., servers), getting back this information used to be very challenging. Today, however, it is not a problem for a number of client targets. Web browsers, PDF viewers and office applications run user controlled scripts (JavaScript, ActionScript, VBScript), which can be used to dynamically construct exploit payloads at run-time on the target machine. Table I lists some recent exploits published by VUPEN [28], which use information leaks and ROP to bypass ASLR and W⊕X. In all examples, the control flow is hijacked at an indirect call instruction (after corrupting a function pointer or the vtable), so stack cookies are not an issue. In all cases, the address leakage is done by exploiting an arbitrary memory write in order to corrupt another pointer which is read later (the fourth column gives more hints). One common way of leaking out memory contents is by overwriting the length field of a (e.g., JavaScript) string object before reading it out (in the user script). As shown in the last column, in case of browser targets, user scripting is used to leak current addresses and to construct the exploit, while in case of ProFTPD, the leaked information is sent back on the network by corrupting a pointer to a status message.

## IV. Approaches and evaluation criteria

The previously identified protection techniques can be divided into two main categories: probabilistic and deterministic protection. Probabilistic solutions, e.g., Instruction Set Randomization, Address Space Randomization, or Data Space Randomization, build on randomization or encryption. All other approaches enforce a deterministic *safety policy* by implementing a low-level reference monitor [29]. A reference monitor observes the program execution and halts it whenever it is about to violate the given security policy. Traditional reference monitors enforce higher level policies, such as file system permissions, and are implemented in the kernel (e.g., system calls).

Reference monitors enforcing lower level policies, e.g., Memory Safety or Control-flow Integrity, can be implemented efficiently in two ways: in hardware or by embedding the reference monitor into the code. For instance, the W⊕X policy (Code Integrity and Non-executable Data) is now enforced by the hardware, as modern processors support both non-writable and non-executable page permissions. Hardware support for a security policy results in negligible overhead. The alternative to hardware support is adding the reference monitor *dynamically* or *statically* to the code.

Since adding new features to the hardware is unrealistic, from this point we focus only on solutions which transform existing programs to enforce various policies. *Dynamic (binary) instrumentation* (e.g., Valgrind [30], PIN [31], DynamoRIO [32], or libdetox [33]) can be used to dynamically insert safety checks into unsafe binaries at runtime. Dynamic binary instrumentation supports arbitrary transformations but introduces some additional slowdown due to the dynamic translation process. Simple reference monitors, however, can be implemented with low overhead (e.g., a shadow stack costs less than 6.5% performance for SPEC CPU2006 in [33]). More sophisticated reference monitors like taint checking [34] or ROP detectors [35] result in overheads that exceed 100% and are unlikely to be deployed in practice. *Static instrumentation* inlines reference monitors statically. This can be done by the compiler or by *static binary rewriting*. Inline reference monitors can implement any safety policy and are usually more efficient than dynamic solutions, since the instrumentation is not carried out at run-time.

Next, we discuss the main properties and requirements for solutions enforcing low-level policies. These properties determine the practicality of a proposed method, more precisely, whether or not it is suitable for wide adoption. We set up our requirements for practicality while discussing a given property.

### A. Protection

**Enforced policy.** The *strength* of the protection is determined by the policy it enforces. The exact policy that a solution enforces determines its *effectiveness*. Different techniques enforce different types of policies (e.g., Memory Safety or Data-flow Integrity) and at different levels. The practical utility of a policy can be described by the attacks it can protect against, out of the four we have identified. Subtle differences in the policies allow or prevent individual attacks. The *accuracy* of an approach is determined by the relation between false negatives and false positives.

| CVE ID | Software | Vulnerability | Address leakage | User scripting |
|---|---|---|---|---|
| CVE-2011-0609 | Adobe Flash | JIT type confusion | Read an IEEE-754 number | ActionScript |
| CVE-2012-0003 | Windows Multimedia Library (affecting IE) | Heap buffer overflow | Read a string after overwriting its length | JavaScript |
| CVE-2011-4130 | ProFTPD | Use-after-free | Overwrite the "226 Transfer Complete" message | none |
| CVE-2012-0469 | Mozilla Firefox | Use-after-free | Read a string after overwriting its length | JavaScript |
| CVE-2012-1889 | Microsoft Windows XML Core Services (affecting IE) | Uninitialized pointer | Read as a RGB color | JavaScript |
| CVE-2012-1876 | Internet Explorer 9/10 (Pwn2Own 2012) | Heap buffer overflow | Read a string after overwriting its length | JavaScript |

Table I
EXPLOITS THAT DEFEAT BOTH DEP AND ASLR USING ROP AND INFORMATION LEAKS

**False negatives.** The possibility of false negatives (protection failures) depends on the definition of the policy. For probabilistic approaches, the probability of a successful attack is always $> 0$ while it is $\geq 0$ for deterministic solutions. As shown in Section III, secrets that the protection relies upon can not only be guessed, but also leaked.

**False positives.** The avoidance of false alarms (e.g., unnecessary crashes) is a very strict requirement for any practical solution. Causing faults in normal operation is unacceptable in production environments. In addition, compatibility issues should not cause any false alarms.

### B. Cost

**Performance overhead.** The cost of a solution is primarily determined by the performance overhead it introduces. Beside security, the most important requirement is speed.

To measure performance, both CPU-bound and I/O-bound benchmarks can be used. CPU-bound benchmarks, such as SPEC [36], are more challenging, because I/O-bound programs spend more time in the kernel, relatively reducing the impact of the user-space CPU overhead. Although some proposals report good scores with selected benchmark programs or with I/O-bound server applications, their overheads are much higher if measured using CPU-bound benchmarks. We recommend that protection approaches considered for wide adoption target CPU-bound client-side programs as well, these being primary targets of today's attacks.

Our comparison analysis in Section IX shows that techniques introducing an overhead larger than roughly 10% do not tend to gain wide adoption in production environments. Some believe the average overhead should be less than 5% in order to get adopted by industry, e.g., the rules of the Microsoft BlueHat Prize Contest [37] confirm this viewpoint.

**Memory overhead.** Inline monitors often introduce and propagate some kind of metadata, which can introduce significant memory overhead as well. Some protection mechanisms (especially the ones using shadow memory) can even double the space requirement of a program. In case of most applications, however, this is much less of an issue than runtime performance.

### C. Compatibility

**Source compatibility.** An approach is source compatible (or source agnostic) if it does not require application source code to be manually modified to profit from the protection. The necessity of even minimal human intervention or effort makes a solution not only unscalable, but too costly as well. Most experts from the industry consider solutions which require porting or annotating the source code impractical.

**Binary compatibility.** Binary compatibility allows compatibility with unmodified binary modules. Transformed programs should still link with unmodified libraries. Backward compatibility is a practical requirement to support legacy libraries. Using unprotected libraries may leave parts of the program exploitable, but allows incremental deployment. Also, for instance on the Windows platform, system libraries are integrity protected and thus cannot be easily changed.

**Modularity support.** Support for modularity means that individual modules (e.g. libraries) are handled separately. A compiler based solution should support separate compilation of modules, while a binary rewriter should support hardening each file (main executable or library) separately. Because dynamic-link libraries (.dll and .so) are indispensable for modern operating systems, all practical protections must support them as well.

## V. PROBABILISTIC METHODS

Probabilistic methods rely on randomization and secrets. There are three main approaches: *Instruction Set Randomization*, *Address Space Randomization*, and *Data Space Randomization*. Figure 1 shows that Instruction Set Randomization (ISR) [38] mitigates attacks based on code corruption and injection of shellcode. Code corruption is prevented by read-only page permissions, and shellcode injection is prevented by non-executable page permissions. Due to hardware improvements, ISR has become obsolete.

Address Space Randomization (ASR) mitigates control-flow hijacking attacks by randomizing the location of code and data and thus the potential payload address. Data Space Randomization (DSR) probabilistically mitigates *all* attacks by randomizing (encrypting) the contents of the memory.

### A. Address Space Randomization

Address Space Layout Randomization (ASLR) [5], [39] is the most prominent memory address randomization technique. ASLR randomly arranges the position of different code and data memory areas. If the payload's address in the virtual memory space is not fixed, the attacker is unable to divert control-flow reliably. ASLR is the most comprehensive currently deployed protection against hijacking attacks.

The diverted jump target can be some injected payload in a data area or existing code in the code section. This is why every memory area must be randomized, including the stack, heap, main code segment, and libraries. The protection can always be bypassed if not *all* code and data sections are randomized. On most Linux distributions, for instance, only library code locations are randomized but the main module is at a fixed address. Most programs are not compiled as Position Independent Executables (PIE) to prevent a $10\%$ on average performance degradation [40].

Furthermore, on 32 bit machines the maximum possible entropy allowed by the virtual memory space is ineffective against brute-force or de-randomization attacks [41]. De-randomization is often carried out by simply filling the memory with repeated copies of the payload, which is called heap-spraying or JIT-spraying [14], [42]. Another potential attack vector is partial pointer overwrites. By overwriting the least significant byte or bytes of a pointer, it can be successfully modified to point to a nearby address [43].

Even if everything is randomized with very high entropy (e.g., on x64 machines), information leaks can completely undermine the protection. Information leaks are the primary attack vector against probabilistic techniques, and as Figure 1 shows, they are always possible if (some level of) Memory Safety is not enforced.

Since the wide deployment of W$\oplus$X the focus of randomization has become code. As illustrated by Step 6 of Figure 1, code reuse attacks became the primary threat. To increase the entropy in code locations, researchers proposed the permutation of functions [44] and instructions inside functions [45] as well. Self-Transforming Instruction Relocation (STIR) [46] randomly re-orders the basic blocks of a binary at launch-time. While these techniques make ROP attacks harder, they usually do not protect against return-to-libc attacks. These techniques also assume that a code reuse (ROP) exploit needs several gadgets, in which case the provided entropy is high enough. However, sometimes a single gadget is enough to carry out a successful attack. The address of a single instruction, gadget, or function is relatively easy to acquire via an information leak.

A technique in the border-land between Address Space and Data Space Randomization is pointer encryption. Cowan et al. [47] proposed PointGuard, which encrypts all pointers in memory and only decrypts them right before they are loaded into a register. This technique can be considered the dual of ASLR, since it also introduces entropy in addresses, but in the "data space": it encrypts the stored address, i.e., pointers' values. To encrypt the pointers PointGuard uses the XOR operation with the same key for all pointers. Since it used only one key, by leaking out one known encrypted pointer from memory, the key can be easily recovered [12]. However the primary reason what prevented PointGuard from wide adoption was that it was neither binary nor source code compatible.

### B. Data Space Randomization

*Data Space Randomization* (DSR) [48] was introduced by Bhatkar and Sekar to overcome the weaknesses of PointGuard and to provide stronger protection. Similarly to PointGuard, DSR randomizes the *representation* of data stored in memory, not the location. It encrypts all variables, not only pointers, and using different keys. For a variable $v$, a key or mask $m_v$ is generated. The code is instrumented to mask and unmask variables when they are stored and loaded from memory. Since several variables can be stored and loaded by the same pointer dereference, variables in equivalent "points-to" sets have to use the same key. The computation of these sets requires a static pointer analysis prior to the instrumentation. The protection is stronger, because encrypting all variables not only protects against control-flow hijacks, but also data-only exploits. Also, the use of multiple keys prevents the trivial information leak described in PointGurad's case, but not in all cases [12].

The average overhead of DSR is 15% on a custom benchmark. The solution is not binary compatible. Protected binaries will be incompatible with unmodified libraries. Also, whenever points-to analysis is needed, modularity will be an issue. Different modules cannot be handled separately, because the points-to graph has to be computed globally. To overcome this issue the authors propose computing partial points-to graphs for separate modules and leave the computation of the global graph to the dynamic linker.

## VI. Memory Safety

Enforcing *Memory Safety* stops all memory corruption exploits. For complete Memory Safety, both spatial and temporal errors must be prevented without false negatives. Type-safe languages enforce both spatial and temporal safety by checking object bounds at array accesses and using automatic garbage collection (the programmer cannot destroy objects explicitly). Our focus is transforming existing unsafe code to enforce similar policies by embedding low-level reference monitors. The instrumentation may be in the source code, intermediate representation, or binary level.

### A. Spatial safety with pointer bounds

The only way to enforce complete spatial safety is to keep track of pointer bounds (the lowest and highest valid address it can point to). CCured [49] and Cyclone [50] use "fat-pointers" by extending the pointer representation to a structure which includes the extra information. Unfortunately these systems need source-code annotations and are therefore impractical for large code bases. Furthermore, changing the pointer representation changes the memory layout, which breaks binary compatibility.

SoftBound [51] addresses the compatibility problem by splitting the metadata from the pointer, thus the pointer representation remains unchanged. A hash table or a shadow memory space is used to map pointers to the metadata. The code is instrumented to propagate the metadata and to check the bounds whenever a pointer is dereferenced. For new pointers, the bounds are set to the starting and ending address of the object it is pointed to. Runtime checks at each pointer dereference ensure that the pointer stays inside bounds. These checks stop all *spatial* errors in the second step of our exploit model.

Pointer based bounds checking is capable of enforcing spatial safety completely without false positives or false negatives if and only if every module is protected. SoftBound is formally proven to provide complete spatial violation detection. Unfortunately, the performance overhead of SoftBound is high, 67% on average. While pointer based approaches, e.g., SoftBound, provide a limited compatibility with unprotected libraries, full compatibility is hard to achieve. Consider, for instance, a pointer created by the protected module. If that pointer is modified by an unprotected module, the corresponding metadata is not updated, causing false positives. We summarize the properties of the main approaches we cover at the end of the paper in Table II.

### B. Spatial safety with object bounds

Because of the compatibility issues caused by pointer based approaches, researchers proposed object based alternatives. Instead of associating the bounds information with pointers, these systems associate the information with the objects. Knowing only the bounds of allocation regions is not enough information to catch errors at pointer dereferences, because we do not know if the pointer points to the *right* object. Hence, object based techniques focus on pointer arithmetic (Step 1 in the model) instead of dereferences (Step 2) to protect the bounds of pointers. Binary compatibility is possible because the metadata is only updated at object creation and deletion. Consider the previous example. The metadata this time is associated with the object rather than with the pointer. If a pointer is updated in an unprotected module, then the metadata will not go out-of-sync.

One problem with this approach, however, is that pointers can legitimately go out of bounds as long as they are not dereferenced. For instance, during the last iteration of a loop over an array, a pointer typically goes off the array by one, but it is not dereferenced. The first binary compatible object based solution to enforce spatial safety is a GCC patch by Jones and Kelly (J&K) [52], which solved this problem by padding allocated objects with an extra byte. This still caused false alarms when a pointer legitimately went out of bounds more than one byte. A more generic solution to this problem was later provided by CRED [53].

The main problem with object based approaches is that they cannot provide complete spatial safety. False negatives can occur, because memory corruption inside objects or structures remains undetected. This is because the C standard allows pointer arithmetic within `struct` fields. E.g., for `memset(&strct,0,sizeof(strct));` the pointer needs to be allowed to iterate through the whole structure.

J&K suffers a large performance overhead of 11-12x. CRED decreased this overhead to around 2x, but by reducing the checked data structures to character arrays only. Dhurjati et al. [54] extend J&K's work by building on a technique called "automatic pool allocation" [55]. Automatic pool allocation partitions the memory based on a static points-to analysis. Partitioning allows using a separate and much smaller data structures to store the bounds metadata for each partition, which can decrease the overhead further to around 120%.

Baggy Bounds Checking (BBC) [56] is currently one of the fastest object based bounds checkers. BBC trades memory for performance and adds padding to every object so that its size will be a power of two and aligns their base addresses to be the multiple of their (padded) size. This property allows a compact bounds representation and an effective way to look up object bounds. The authors of BBC claim that their solution is around twice as fast than the previously mentioned Dhurjati's automatic pool allocation based optimization. BBC's average performance overhead is 60% on the SPECINT 2000 benchmark. PAriCheck [57] was developed concurrently with BBC. It pads and aligns objects to powers of two as well for efficient bounds checking. It has slightly better performance cost and memory overhead than BCC.

The motivation for object based approaches is to remain compatible with unprotected libraries to reduce false positives. If an allocation or de-allocation happens in an unprotected library, the metadata is set by intercepting `malloc` and `free`. For every other object created in an unprotected library, default values are used, allowing arbitrary arithmetic.

### C. Temporal safety

Spatial safety alone does not prevent all vulnerabilities. Use-after-free and double-free vulnerabilities remain undetected by the previously discussed bounds checkers. Numerous approaches have been proposed to enforce temporal safety.

*1) Special allocators:* The naïve approach to protect against use-after-free exploits would be to never reuse the same virtual memory area, but that would be overly wasteful. Special memory allocators, like Cling [58], are designed to thwart dangling pointer attacks without significant memory or performance overhead. Cling is a replacement for `malloc`, which allows address space reuse only among objects of the same type and alignment. This policy does not prevent dereferences through dangling pointers, but enforces type safe memory reuse, preventing the described use-after-free attack. Dynamic memory allocator replacements of course cannot prevent unsafe reuse of local, stack allocated objects.

*2) Object based approaches:* Perhaps the most widely used tools to detect memory errors in practice is Valgrind's Memcheck [30] tool and AddressSanitizer [59]. These tools try to detect use-after-free bugs by marking locations which were de-allocated in a shadow memory space. Accessing a newly de-allocated location can be detected this way. This approach, however, fails to detect errors after the area is re-allocated for another pointer: the area is registered again and the invalid access remains undetected. The object based bounds checkers described in the previous subsection offer the same protection, since de-allocation invalidates the object in the metadata table. Valgrind, being a dynamic translator, causes a 10x slowdown in average, while AddressSanitizer causes 73% slowdown by instrumenting code at compile time. The only way to detect a use-after-free attack reliably is to associate the temporal information with the pointer and not with the object.

*3) Pointer based approaches:* Maintaining not only bounds but also allocation information with pointers allows enforcing full Memory Safety. Allocation information tells if the pointed to object is still valid. It is not enough to just keep an extra bit associated with each pointer indicating the object's validity, because all pointers pointing to it have to be found and updated when the object is freed. CETS [60] extends SoftBound and solves the problem by eliminating the redundancy of the above described naïve idea. The validity bit is stored only at one place in a global dictionary. Each new object gets a unique identifier used as the key to the dictionary and pointers are associated with this unique ID. A special data structure for the dictionary allows the quick and easy invalidation of objects and also fast lookups to check object validity. CETS is formally proven to enforce temporal safety, if spatial safety is also enforced. In other words, together with SoftBound, CETS enforces Memory Safety. The average execution overhead of the instrumentation enforcing temporal safety alone is 48%. When coupled with SoftBound to enforce complete Memory Safety, the overhead is 116% on average on the SPEC CPU benchmark. As a pointer based solution, CETS suffers the same binary compatibility issues as SoftBound when it comes to unprotected libraries.

## VII. Generic attack defenses

*Data Integrity* and *Data-flow Integrity* are weaker policies than *Memory Safety*. They aim to protect against both control data (hijacking) and non-control data attacks, but not against e.g., information leaks. While the former policy prevents data corruption, the latter detects it.

### A. Data Integrity

Data Integrity solutions enforce an approximation of spatial memory integrity. These techniques focus on the most common attacks, which start by writing through an out of bounds pointer. They do not enforce temporal safety, and they only protect against invalid memory writes, not reads. Furthermore, they only approximate the spatial integrity enforced by the previously covered bounds checkers in order to minimize the performance overhead. In all cases the approximation is due to a static pointer analysis carried out prior to the instrumentation.

*1) Integrity of "safe" objects:* The technique proposed by Yong et al. [61] first identifies the subset of "unsafe pointers". A pointer is considered unsafe if it might go out of bounds, e.g., because it is a computed value (`p[i]`). A static pointer analysis identifies the unsafe pointers together with their points-to sets, i.e., their potential target objects. Let us call the union of the identified points-to sets *unsafe objects*. The code is instrumented to mark each byte of an unsafe object in a shadow memory area at its creation and clear it at deallocation. Checks are inserted before each write dereference of an unsafe pointer to check whether the location is marked in the shadow memory. This prevents the corruption of any data in a memory area that does not belong to an unsafe object.

This policy is sufficient to protect not only variables which are never accessed through pointers, but, for instance, saved return addresses as well. However, sensitive variables can still be identified as unsafe objects and thus remain unprotected. The authors also mention that out of 101 function pointers in their benchmark suite, two ended up in the set of unsafe objects, which means that in case of a memory error, these values can be corrupted. Since reads are left unchecked, any value can be corrupted when read into a register via a bad pointer. This allows certain control-flow hijack attacks, program specific attacks, and information leak attacks as well.

The reported runtime overhead of Yong's system varies between 50-100% on the SPEC 2000 benchmark. Uninstrumented libraries raise compatibility issues. If a pointer is dereferenced in the transformed module, accessing an object created by an unprotected module, then a false alarm is triggered. This problem can be mitigated in case of heap objects by wrapping memory allocating functions to mark every allocated area in the shadow memory.

*2) Integrity of points-to sets:* The previous technique restricts pointer dereferences to write only unsafe object. Write Integrity Testing (WIT) [62] further strengthens the above policy by restricting each pointer dereference to write only objects in its own points-to set. Naturally, the pointer analysis only results in a conservative approximation of the set of objects a pointer may point to. The calculated distinct points-to sets are associated with different ID numbers, which are used to mark the objects in the shadow memory area. While Yong's approach only uses two IDs: 1 for unsafe objects and 0 for everything else, WIT marks objects belonging to separate points-to sets with different IDs.

Furthermore, WIT checks indirect calls as well to stop hijacking attacks, which the previous policy left possible. It protects indirect calls by calculating the points-to set of pointers used by indirect call instructions and associate them with IDs as well. The IDs are placed in the shadow memory for valid code target addresses and, as in case of indirect writes, before each indirect call the IDs are checked. Functions associated with the same ID are still interchangeable.

The policy enforced by WIT is stronger than Yong's approach due to distinguishing different points-to sets, but objects assigned to the same ID remain vulnerable. Since WIT does not protect reads, data can be corrupted when read into a register, and information leaks are possible as well. Due to the missing read checks function pointers can be corrupted, too. This is why WIT checks indirect calls instead in order to detect the corruption. Checking the target of indirect control transfers makes WIT a *Control-flow Integrity* approach, which is covered in Section VIII-B. Notice that while calls are checked, returns are not. This is because returns can only be corrupted via writes, as they are never read by dereferencing a pointer, and thus they are considered protected. Since WIT does not deal with temporal errors either, overwriting a return address via an escaped dangling pointer is still possible, however such bugs are rare in practice.

The reported performance overhead of WIT is around 5-25% for the SPEC benchmark. The approach is not binary compatible. Using uninstrumented libraries can create false alarms, because they do not maintain the object IDs at allocations. Like in case of DSR, or other solutions dealing with distinct points-to sets, modularity is also an issue, since the resulting IDs depend on the global points-to graph. While WIT works at compile time, BinArmor [63] aims to enforce a similar policy with binary rewriting. Since the pointer analysis the policy requires is infeasible to do in binaries, the system tries to identify potentially unsafe dereferences and their valid targets dynamically, by running and tracing the program with various inputs. This approach can neither guarantee the lack of false negatives, nor false positives, and its performance overhead can go up to 180%.

### B. Data-flow Integrity

*Data-Flow Integrity* (DFI) as proposed by Castro et al. [64] detects the corruption of any data before it gets used by checking read instructions. DFI restricts reads based on the last instruction that wrote the read location. In program analysis terms, DFI enforces the reaching definition sets. The reaching definition set of an instruction is the set of instructions which might have last written (defined) the value that is used by the given instruction based on the control-flow graph. For instance, the policy ensures that the `isAdmin` variable was last written by the write instruction that the source code defines and not by some rogue attacker-controlled write. Or it ensures that the return address used by a return was last written by the corresponding call instruction. DFI also builds on static points-to analysis in order to compute the global reaching definition sets. Similarly to WIT, the resulting reaching definition sets are assigned a unique ID. Each written memory location is marked in the shadow memory with the writing instruction's ID. Before each read, this ID is checked whether it is the element of the statically computed set of allowed IDs.

The previously discussed solutions checked every indirect memory write, so the shadow memory area was automatically protected. Contrarily, the Data-flow Integrity policy dictates the instrumentation of only reads. Unfortunately, in order to protect the integrity of its metadata, DFI has to check all indirect writes as well, and make sure that their target addresses are outside of the shadow memory area.

The performance overhead of the technique varies between 50-100% on the SPEC 2000 benchmark. Similarly to previous solutions, it is not binary compatible, since false alarms can be caused by the lack of metadata maintenance in unprotected libraries.

## VIII. Control-flow hijack defenses

The following two policies focus only on hijacking attacks. While the *Code Pointer Integrity* aims to prevent the corruption of code pointers, *Control-flow Integrity* detects it.

### A. Code Pointer Integrity

While the integrity of some code pointers can and should be protected, enforcing Code Pointer Integrity alone is infeasible. Immutable code pointers, such as the ones in the Global Offset Table or in virtual function tables (vtable), can be easily protected by keeping them in read-only memory pages. Most code pointers however, such as programmer defined function pointers or saved return addresses, must remain writable. Furthermore, even if the integrity of all code pointers in memory could be enforced, the hijacking attack would still be possible, by exploiting an erroneous indirect memory read to load the wrong value into the register. Most use-after-free exploits, for instance, divert the control-flow by reading the "wrong" virtual function table through a dangling pointer, which does not involve

overwriting code pointers in memory at all. It follows from this discussion that detecting a code pointer corruption before its usage would be better.

### B. Control-flow Integrity

Control-flow Integrity (CFI) solutions enforce some policy regarding indirect control transfers, mitigating the hijacking attack in Step 5. Note that direct control transfers cannot be diverted, and hence they need no protection.

*1) Dynamic return integrity:* The most well known control-flow hijacking attack is the "stack smashing" attack [65]. Stack smashing exploits a buffer overflow in a local variable to overwrite the return address on the stack. Stack cookies or canaries [66] are the first proposed solution against this attack. A secret value (cookie/canary) is placed between the return address and the local variables. If the return address is overwritten by a buffer overflow, the cookie changes as well, what is detected by the check placed before the return instruction. Stack cookies do not protect indirect calls and jumps, and they are vulnerable to direct overwrite attacks and information leaks. However, stack cookies are popular and widely deployed, because the performance overhead is negligible (less than 1%) and no compatibility issues are introduced.

Shadow stacks [67] can solve some of the problems of canaries, like information leaks and direct overwrites. To eliminate the reliance on a secret, the saved return addresses are pushed to a separate shadow stack as well, so upon function return, the shadow copy can be compared with the original return address. Simply making a copy and checking if it still matches before the return makes the attack much harder, even when the shadow stack is not protected, since the attacker has to corrupt the return address in two separate locations. To protect the shadow stack itself, RAD [68] proposes the use of guard pages or switching write permission to protect the shadow stack area. While the former does not protect against direct overwrites, the latter causes 10x slowdown. To estimate the performance overhead of an unprotected shadow stack mechanism, we implemented one as an LLVM plugin, which has an average overhead of 5% on the SPEC2006 benchmark. Shadow stack mechanisms also has to deal with compatibility issues, e.g., to handle exceptions. However, we believe that false positives can be avoided by a careful implementation.

*2) Static control-flow graph integrity:* To prevent all control-flow hijacks, not only returns, but indirect calls and jumps have to be protected as well. Section VII covers how WIT identifies and enforces the set of valid targets (i.e., the points-to set) of each call instruction. This idea, together with the term Control-flow Integrity was originally introduced by Abadi et al. [69]. Their work focuses on statically determining the valid targets of not only calls, but also function returns, and thus enforcing the resulting static control-flow graph. Unlike WIT, which stores the IDs in a protected shadow memory, the CFI authors propose storing them inside the code itself, by placing the ID right to the target location, so it can be protected by *Code Integrity*. To avoid compatibility issues, the IDs can be encoded into instructions, which, if inserted, will not affect the semantics of the code. Calls and returns are instrumented to check the target address whether it has the right ID before jumping there. Note, that this requires *Non-executable Data* as well, to prevent forging valid targets.

As for returns, enforcing any statically predetermined set of valid targets is a weaker policy than enforcing the dynamic call stack enforced by a shadow stack. At run-time, there is always exactly one correct target of a function return, but since a function can be called from multiple call sites, the statically determined set will include all of them as valid targets.

Another issue with enforcing the unique points-to sets of indirect control transfers is modularity support, as in the case of all previously covered pointer analysis based solutions. The precise points-to sets can only be determined globally, which makes modularity and dynamic library reuse challenging. This the main reason why this solution works great with monolithic kernels [70] or hypervisors [71], where every module is statically linked together, but has not been deployed for dynamically linked applications. A weaker, but more practical policy is restricting indirect control transfers to the union of all their points-to sets (cf. Yong et al. in Section VII-A). The original CFI implementation also uses this approach, meaning that all indirectly callable function is marked by the same ID. The advantage of this policy is that it does not even need pointer analysis, because it is enough to enumerate all functions whose address is taken. This is a much more conservative policy, but it allows the modular transformation and interchanging of libraries. For many functions, this policy means that the allowed set of return targets has to include all call sites in a program. Since this is overly permissive, the authors suggest using a shadows stack mechanism instead for checking returns.

The average performance overhead of the Abadi implementation is 15%, while the maximum measured is as high as 45%. The implementation which uses a shadow stack mechanisms for returns has an additional 10% overhead. This solution is not binary compatible either, and since it relies on the W$\oplus$X policy, it can not be enforced in case of JIT compilation.

## IX. Discussion

We summarize the properties of a selected set of solutions in Table II, grouped by the policy categories identified in Section II, except the following two: (i) Code Pointer Integrity, because enforcing it is infeasible without any level of Memory Safety; (ii) Instruction Set Randomization, because the same attack vectors can be defeated with page permission enforcing Code Integrity and Non-executable

| | Policy type (main approach) | Technique | Perf. % (avg/max) | Dep. | Compatibility | Primary attack vectors |
|---|---|---|---|---|---|---|
| **Generic prot.** | Memory Safety | SofBound + CETS | 116 / 300 | × | Binary | — |
| | | SoftBound | 67 / 150 | × | Binary | UAF |
| | | Baggy Bounds Checking | 60 / 127 | × | — | UAF, sub-obj |
| | Data Integrity | WIT | 10 / 25 | × | Binary/Modularity | UAF, sub-obj, read corruption |
| | Data Space Randomization | DSR | 15 / 30 | × | Binary/Modularity | Information leak |
| | Data-flow Integrity | DFI | 104 / 155 | × | Binary/Modularity | Approximation |
| **CF-Hijack prot.** | Code Integrity | Page permissions (R) | 0 / 0 | ✓ | JIT compilation | Code reuse or code injection |
| | Non-executable Data | Page permissions (X) | 0 / 0 | ✓ | JIT compilation | Code reuse |
| | Address Space Randomization | ASLR | 0 / 0 | ✓ | Relocatable code | Information leak |
| | | ASLR (PIE on 32 bit) | 10 / 26 | × | Relocatable code | Information leak |
| | Control-flow Integrity | Stack cookies | 0 / 5 | ✓ | — | Direct overwrite |
| | | Shadow stack | 5 / 12 | × | Exceptions | Corrupt function pointer |
| | | WIT | 10 / 25 | × | Binary/Modularity | Approximation |
| | | Abadi CFI | 16 / 45 | × | Binary/Modularity | Weak return policy |
| | | Abadi CFI (w/ shadow stack) | 21 / 56 | × | Binary/Modularity | Approximation |

Table II
THIS TABLE GROUPS THE DIFFERENT PROTECTION TECHNIQUES ACCORDING TO THEIR POLICY AND COMPARES THE PERFORMANCE IMPACT, DEPLOYMENT STATUS (DEP.), COMPATIBILITY ISSUES, AND MAIN ATTACK VECTORS THAT CIRCUMVENT THE PROTECTION.

data. The comparison neither covers overly specific Data Integrity protections like heap metadata protection or canary/redzone/guards based protections, nor too special cases of Control-flow Integrity, like exception handler validation. It does not include dynamic binary instrumentation solutions due to their high performance cost, or others which are not fully automatic (e.g., needs source code modification). The upper half of the table covers protections aiming to protect against memory corruption in general and thus mitigate all four different attacks identified in Section II. The lower half covers approaches which aim to protect against control-flow hijacks only.

The performance is represented as the average and maximum overhead using either the SPEC CPU 2000 or 2006 benchmarks. We rely on the numbers reported by the developers of the tools, since several of them are not publicly available. We stress that since the values represent measurements in different environments, different configurations, and sometimes with different sets of programs, they only provide rough estimates. We present some of the fastest solutions for enforcing Memory Safety and Data-flow Integrity but even those can double the execution time. WIT and DSR report much lower overhead then other general protection techniques and even smaller then the Abadi CFI system under the hijacking protections. The deployment status column represents whether a solution is used in practice. The case of enforcing full ASLR on Linux shows that even a 10-25% overhead prevents deployment. This is the overhead that Position Independent Executables (relocatable executables) cause on 32-bit machines and the reason why ASLR is enforced only for libraries by default on most distributions. As the table shows, only solutions with negligible overhead are adopted in practice.

Not only performance but also compatibility issues prevent the deployment of many proposed techniques. In the table, "binary" represents binary compatibility issues, meaning interfacing with unmodified binaries (e.g., legacy libraries). This will not only cause false negatives, but false positives as well, which contradicts with our practical requirements. All solutions which build on points-to analysis have "modularity" issues, because the enforcement of stricter policies require consideration of the dependencies between modules, which makes re-usable libraries challenging.

None of the shown policies are perfect regarding robustness, except enforcing complete Memory Safety with pointer based techniques. Protections enforcing weaker Memory Safety policies have more attack vectors, like use-after-free (UAF), corruption of sub-objects (sub-obj) or corrupting values in registers via read dereferences. DSR and ASLR provide the most comprehensive solutions as a generic and hijacking protection respectively, but both of them can be circumvented by information leaks. The protection level of DFI (as a generic protection) and CFI (as a hijack protection) is only bounded by the "approximation" due the static analysis. This means (i) enforcing a static set of valid reaching definitions or jump targets, and not the single dynamically valid ones, and (ii) the conservativeness of the analysis establishing those sets. WIT is shown twice in the table as a generic protection and as a control-flow hijacking protection. As a generic protection, it enforces an "approximation" of Memory Safety, i.e., Data Integrity, which has some weaknesses, but protects against most of the attacks, including program specific data-only attacks as well. As a control-flow hijacking protection, it enforces the same policy as Abadi CFI with a shadows stack, but with less overhead. Unfortunately, neither the Abadi CFI nor the WIT approach has been adopted in practice, although their overhead can be considered low. We attribute this to the beforementioned compatibility and modularity problems raised by points-to analysis.

## X. Conclusion

Both academia and industry have been fighting memory corruption bugs for decades. From time to time, pundits have proclaimed that the problem had finally been solved, only to find their assertions falsified by subsequent attacks. With the wide deployment and hardware support for Non-executable Data, research has been focusing on ROP attacks within hijack protections. The latest solution seemed to be randomization, such as fully enforced, high entropy, 64-bit ASLR or other in-place randomization techniques. But the increased use of JIT compilation limits the usability of a W$\oplus$X policy, while the prevalence of user scripting simplifies defeating randomization. The ability of running attacker provided scripts helps leaking secrets and on-the-spot dynamic exploit construction. Researchers have to step back, and instead of focusing on specific attacks, we need to look at the big picture. Hopefully, this paper helps in this regard by setting up its general attack model and by placing different policy types in this model.

This systematization suggests that stronger policies are needed, such as Data Integrity; or, when only hijacking attacks are considered a valid threat, Control-flow Integrity. While the research direction of enforcing such policies is promising, existing solutions are still impractical. Our requirement analysis and the summarization of current techniques show that performance, and especially compatibility problems, are the main barriers of wide adoption. We remind researchers in the security area to recognize the significance of these properties in the real world.

There is a pressing need for research, development, and deployment of better publicly available software protection techniques, especially built into commonly used compilers, such as LLVM and GCC. These open-source platforms can be of great value, where some of the compatibility problems can be solved by the community so researchers can release their robust but possibly slow protections to interested users. Such experiments interacting mutually with real applications will improve research further, and they might be able to lift the performance threshold people impose on security. We hope that this systematization of knowledge will help other researchers in finding new ways to make progress in this area. The war is not over.

**Acknowledgments.** We thank the anonymous reviwers, Prof. R. Sekar, Stephen McCamant, and Dan Caselden for their insightful reviews, helpful comments and proof-reading. This work was supported in part by ONR grants N000140710928 and N000140911081; NSF grants CNS-0831298, 0842695, 0831501 and CCF-0424422; an AFOSR grant FA9550-09-1-0539; a DARPA award HR0011-12-2-005; and the National Natural Science Foundation of China grant No. 61003216.

## References

[1] MITRE, "CWE/SANS Top 25 Most Dangerous Software Errors," 2011, http://cwe.mitre.org/top25.

[2] H. Etoh and K. Yoda, "Protecting from stack-smashing attacks," 2000, http://www.trl.ibm.com/projects/security/ssp.

[3] Microsoft, "/SAFESEH (Safe Exception Handlers)," 2003, http://msdn2.microsoft.com/en-us/library/9a89h429.aspx.

[4] A. van de Ven and I. Molnar, "Exec shield," 2004, http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf.

[5] T. PaX, "Address space layout randomization," 2001. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[6] Nergal, "The advanced return-into-lib(c) exploits: PaX case study," *Phrack*, vol. 11, no. 58, Dec 2001.

[7] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls," in *CCS'07*.

[8] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *CCS'10*.

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *ASIACCS'11*.

[10] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *RAID'11*.

[11] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Sys. Sec.'12*.

[12] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *EUROSEC'09*.

[13] F. J. Serna, "CVE-2012-0769, the case of the perfect info leak," 2012.

[14] D. Blazakis, "Interpreter exploitation," in *WOOT'10*.

[15] U. Erlingsson, Y. Younan, and F. Piessens, "Low-level software security by example," in *Handbook of Information and Communication Security*, 2010.

[16] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory errors: the past, the present, and the future," in *RAID'12*.

[17] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against C and C++ programs," *ACM Comp. Surv.'12*.

[18] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kguard: lightweight kernel protection against return-to-user attacks," in *USENIX Security'12*.

[19] jp, "Advanced Doug Lea's malloc exploits," *Phrack*, vol. 11, no. 61, Aug 2003.

[20] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: defeating return-oriented programming through gadget-less binaries," in *ACSAC'10*.

[21] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *IEEE SP'12*.

[22] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *SOSP'93*.

[23] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: software guards for system address spaces," in *OSDI'06*.

[24] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy,

S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A sandbox for portable, untrusted x86 native code," in *IEEE SP'09*.

[25] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *USENIX Security'05*.

[26] huku and argp, "Exploiting vlc: A case study on jemalloc heap overflows," *Phrack*, vol. 14, no. 68, Apr 2012.

[27] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *USENIX Securty'11*.

[28] VUPEN, "Vulnerability research team blog," 2012, http://www.vupen.com/blog/.

[29] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Sys. Sec.'00*.

[30] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI'07*.

[31] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI'05*.

[32] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *CGO '03*.

[33] M. Payer and T. R. Gross, "Fine-grained user-space security through virtualization," in *VEE'11*.

[34] E. Bosman, A. Slowinska, and H. Bos, "Minemu: the world's fastest taint tracker," in *RAID'11*.

[35] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: a detection tool to defend against return-oriented programming attacks," in *ASIACCS'11*.

[36] C. D. Spradling, "SPEC CPU2006 benchmark tools," *SIGARCH Comp. Arch. News'07*.

[37] Microsoft, "The BlueHat prize contest official rules," 2012, http://www.microsoft.com/security/bluehatprize/rules.aspx.

[38] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *CCS '03*.

[39] M. Chew and D. Song, "Mitigating Buffer Overflows by Operating System Randomization," Tech. Rep., 2002.

[40] M. Payer, "Too much PIE is bad for performance," 2012.

[41] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *CCS'04*.

[42] T. Wei, T. Wang, L. Duan, and J. Luo, "Secure dynamic code generation against spraying," in *CCS'10*.

[43] T. Durden, "Bypassing PaX ASLR protection," *Phrack*, vol. 11, no. 59, Jul 2002.

[44] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity softwar," in *ACSAC'06*.

[45] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *IEEE SP'12*.

[46] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *CCS'12*.

[47] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard: protecting pointers from buffer overflow vulnerabilities," in *USENIX Security'03*.

[48] S. Bhatkar and R. Sekar, "Data Space Randomization," in *DIMVA'08*.

[49] G. C. Necula, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy code," in *POPL'02*.

[50] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *USENIX ATC'02*.

[51] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for C," *SIGPLAN Not.'09*.

[52] R. Jones and P. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," *Auto. and Algo. Debugging'97*.

[53] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *NDSS'04*.

[54] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *ICSE'06*.

[55] C. Lattner and V. Adve, "Automatic pool allocation: improving performance by controlling data structure layout in the heap," in *PLDI'05*.

[56] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors," in *USENIX Security'09*.

[57] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "PAriCheck: an efficient pointer arithmetic checker for C programs," in *ASIACCS'10*.

[58] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *USENIX Security'10*.

[59] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *USENIX ATC'12*.

[60] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *ISMM'10*.

[61] S. H. Yong and S. Horwitz, "Protecting C programs from attacks via invalid pointer dereferences," in *ESEC/FSE-11'03*.

[62] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *IEEE SP'08*.

[63] A. Slowinska, T. Stancescu, and H. Bos, "Body armor for binaries: preventing buffer overflows without recompilation," in *USENIX ATC'12*.

[64] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *OSDI'06*.

[65] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, Nov. 1996.

[66] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security'98*.

[67] Vendicator, "Stack Shield: A "stack smashing" technique protection tool for linux," 2000.

[68] T. Chiueh and F.-H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *ICDCS'01*.

[69] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS'05*.

[70] J. Li, Z. Wang, T. K. Bletsch, D. Srinivasan, M. C. Grace, and X. Jiang, "Comprehensive and efficient protection of kernel control data," *IEEE Trans. Inf. Forencics and Sec'11*.

[71] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *IEEE SP'10*.