

# Fully Randomized Pointers

GREGORY J. DUCK, National University of Singapore, Singapore  
SAI DHAWAL PHAYE, National University of Singapore, Singapore  
ROLAND H. C. YAP, National University of Singapore, Singapore  
TREVOR E. CARLSON, National University of Singapore, Singapore

Software security continues to be a critical concern for programs implemented in low-level programming languages such as C and C++. Many defenses have been proposed in the current literature, each with different trade-offs including performance, compatibility, and attack resistance. One general class of defense is pointer *randomization* or *authentication*, where invalid object access (e.g., memory errors) is obfuscated or denied. Many defenses rely on the program termination (e.g., crashing) to abort attacks, with the implicit assumption that an adversary cannot “brute force” the defense with multiple attack attempts. However, such assumptions do not always hold, such as hardware speculative execution attacks or network servers configured to restart on error. In such cases, we argue that most existing defenses provide only weak effective security.

In this paper, we propose *Fully Randomized Pointers* (FRP) as a stronger memory error defense that is resistant to even brute force attacks. The key idea is to fully randomize pointer bits—as much as possible while also preserving binary compatibility—rendering the relationships between pointers highly unpredictable. Furthermore, the very high degree of randomization renders brute force attacks impractical—providing strong effective security compared to existing work. We design a new FRP encoding that is: (1) compatible with existing binary code (without recompilation); (2) decoupled from the underlying object layout; and (3) can be efficiently decoded on-the-fly to the underlying memory address. We prototype FRP in the form of a software implementation (BLUEFAT) to test security and compatibility, and a proof-of-concept hardware implementation (GREENFAT) to evaluate performance. We show that FRP is secure, practical, and compatible at the binary level, while a hardware implementation can achieve low performance overheads (<10%).

## 1 INTRODUCTION

Memory errors, such as buffer overflows and use-after-free, are of critical concern for software systems implemented in low-level programming languages such as C and C++. Even today, memory errors continue to be the primary cause of security vulnerabilities—Google and Microsoft report that “70% of our serious security bugs are memory safety problems” [9, 35]. As such, many different tools and techniques for defending against memory errors, in both software and hardware, have been proposed [4, 7, 14–17, 29, 30, 37, 38, 40, 42, 48, 49, 54]. Some tools (e.g., [7, 15, 40, 49]) are primarily designed for *software testing*—i.e., the detection of memory errors during testing/fuzzing. Others (e.g. [4, 17, 42]) are primarily designed for *security hardening*—i.e., preventing memory errors being actively exploited by a proactive attacker. The latter will be the focus of this paper.

Due to various challenges and difficulties, no memory safety hardening solution has enjoyed wide-spread “always-on” adoption. Existing state-of-the-art tools tend to have inherent trade-offs, especially with regard to performance, error coverage, and binary compatibility; and these can hinder real-world application. For program hardening under adversarial attacker models, another important trade-off is *bypass resistance*—i.e., how easily a proactive attacker (who is aware that the program is hardened) can adapt their attack to bypass the defense? Many existing tools tend to have poor resistance. For example, bug detection tools based on *memory poisoning*, such as AddressSanitizer (ASAN) [49] and Valgrind [40], famously do not track pointer origins (a.k.a., *pointer provenance* [34]). While this does achieve good compatibility, it nevertheless allows attackers to construct seemingly valid pointers to out-of-bounds objects [17]—effectively bypassing the defense.

---

Authors’ addresses: Gregory J. Duck, gregory@comp.nus.edu.sg, National University of Singapore, Singapore; Sai Dhawal Phaye, sdp@nus.edu.sg, National University of Singapore, Singapore; Roland H. C. Yap, National University of Singapore, Singapore, ryap@comp.nus.edu.sg; Trevor E. Carlson, tcarlson@comp.nus.edu.sg, National University of Singapore, Singapore.

Other solutions, such as *fat pointers* [39] and *Softbound* [37], do explicitly track pointers with attached object metadata, and this can be more difficult to bypass. However, such tools require recompilation from source code (i.e., no binary compatibility), and use of instrumentation means that this (and other software-based memory safety solutions) tend to incur high performance overheads. Furthermore, since object metadata must be explicitly tracked over function boundaries and to/from memory, these solutions change the *Application Binary Interface* (ABI), meaning that instrumented and uninstrumented binary code cannot be freely mixed. In summary, *existing software-based solutions have poor performance and mixed compatibility/resistance*.

Recently, several hardware-based memory safety solutions, based on pointer *tagging*, *authentication*, or *encryption*, have been proposed. One common idea is to attach a *Pointer Authentication Code* (PAC, e.g., PACMem [30]) or *tag* (e.g., HeapCheck [48]) to each pointer. The tag typically resides in the (otherwise unused) upper pointer bits, and uniquely identifies an underlying object that corresponds to the pointer. When dereferenced, the memory access can be checked against the underlying object bounds, allowing for memory errors to be detected. Another related idea is *Cryptographic Capability Computing* ( $C^3$ ) [29], which partially *encrypts* an encoded pointer value, rather than using separate authentication codes/tags. The encryption hardens against tampering of the relationship between pointers and the underlying object. Being based on hardware extensions, these solutions tend to achieve good binary compatibility and low performance overheads. For security, we can classify these solutions as *entropy*-based defenses, since an attacker must determine a (randomized) tag or encrypted value in order to successfully bypass the defense. Assuming the defense is well designed, the attacker will have no better method other than to guess the correct value, and an incorrect guess will lead to immediate program termination. Each of PACMem, HeapCheck, and  $C^3$  use **24 bits** of entropy, meaning that these provide some baseline resistance to isolated bypass attack attempts.

However, it has been shown that even entropy-based defenses can still be bypassed in practice. The key assumption (that an attacker can only make a single incorrect guess) does not always hold. If this assumption were to be violated, the attacker may simply “brute force” the required value, significantly weakening the bypass resistance of the given defense. For example, it is not uncommon for programs to be configured to automatically restart after a crash, e.g., a network server may automatically restart in order to minimize service disruption. Alternatively, there may be multiple vulnerable servers on a given network. Such configurations effectively grant the attacker multiple attempts [1]. Another attack vector is *side channels*, such as *speculative execution* [28], which allow the attacker to test values without causing program termination. This approach has been proven effective against ARM pointer authentication (PAC), see the PACMan attack [44]. Finally, the attacker may use well-known exploitation techniques, such as *heap spraying*, to improve the probability of making a correct guess. If applicable, heap spraying can be used in combination with the other methods to enhance the overall attack effectiveness. Such bypass attacks are feasible since most existing defenses use low entropy. For example, with 24 bits of entropy, the defense can be bypassed with an average of  $2^{24-1} \approx 8$  million attempts, which is very achievable in practice. In summary, *existing hardware-based solutions have good performance and binary compatibility, but still exhibit some weakness regarding bypass resistance*.

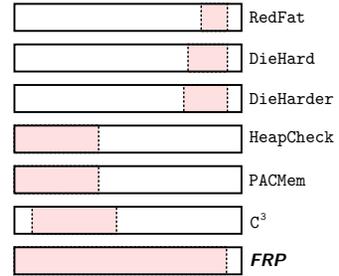


Fig. 1. Relative entropy (position and size) for each defense. More = better.

In this paper, we propose a new hardware-based memory error defense that is designed to *maximize* entropy and therefore maximize bypass resistance. Our approach is based on a simple observation that most (binary) software is agnostic to the underlying pointer encoding—provided that core pointer operations (arithmetic, difference, comparison, dereference, etc.) implement the expected semantics. Thus, we can design a new encoding that represents pointers as *cryptographically secure random numbers*, ensuring that two pointers to different objects will have: (i) bit patterns with no predictable relation to each other; and (ii) no predictable relation to the underlying address(es) in memory. As shown by prior work (e.g., DieHard(er) [4, 42]) randomization protects against memory errors and bypass attacks by making potential targets *unpredictable*—e.g., a buffer overflow is unlikely to find a valid target. However, unlike DieHard(er), our approach randomizes the pointer encoding itself rather than underlying object layout within memory. Our approach works by maintaining an explicit mapping between pointer values and the object they represent. When a pointer is dereferenced, the map is consulted, allowing for the object to be accessed if the pointer is valid (i.e., not out-of-bounds). The use of an explicit map allows pointer values to take randomized values without changing the underlying memory layout. Our approach therefore allows for a significantly higher degree of randomization: as many pointer bits as possible *without breaking compatibility*—a.k.a., *Fully Randomized Pointers* (FRP). For existing x86\_64 binary code, we show that a baseline entropy of **52 bits** can be supported under conservative assumptions, offering robust memory error protection with significantly stronger resistance to bypass attacks ( $\sim 2^{51}$  attack attempts required versus  $\sim 8$  million). Under relaxed assumptions, the effective entropy is further improved up to the full **64 bits**. Our approach is illustrated in Figure 1. Here, each colored inner box represents the position and scale of the bits that the attacker must guess in order to bypass the corresponding defense. This example models 60 bits of randomization, meaning that FRP offers very meaningful resistance against brute force-style attacks.

We have prototyped FRP as a software implementation (BLUEFAT) to test binary compatibility and security, and as a hardware simulation (GREENFAT) to test performance. We show that, despite using a radically different pointer encoding, FRP is compatible with existing x86\_64 binary code without the need for recompilation. In fact, the binary software is even unaware that a memory error defense has even been applied, i.e., our design is fully transparent. We also present (GREENFAT) as a proof-of-concept hardware simulation of FRP, achieving an overhead of ( $<10\%$ ), which is still within the practical range necessary for widespread deployment. Our main contributions are:

- We propose a new pointer encoding that is (1) decoupled from the underlying machine address, and (2) fully randomized. The high degree of randomization hardens against memory errors and brute-force style bypass attacks.
- We show that the randomized pointer design is both efficient and compatible with existing x86\_64 binary code (without recompilation) using dynamic pointer decoding.
- We have implemented FRP in software (BLUEFAT) as well as a hardware simulation (GREENFAT). We show that FRP offers strong memory safety (object-bounds and use-after-free), bypass resistance, and high compatibility. We also show that a hardware implementation of FRP can achieve low performance overheads that are more practical for general use cases.

### Open Source Release (Software)

- <https://github.com/GJDuck/BlueFat>

## 2 BACKGROUND

In this section we briefly summarize the threat model, existing mitigations, the problem statement, and our design.

Table 1. Summary of memory error defenses.

Defense	Methodology		Error Detection					Entropy
	technology	binary? bypass resist.	overflow	underflow	use-after-free	false positives	false negatives	
ASLR	randomization	b ○	○	○	○	-	✗	n.a.
efence [43]	page permissions	b ○	●	●	●	-	✗	n.a.
Valgrind [40]	mem. poisoning	b ○	●	●	●	-	✗	n.a.
DrMemory [7]	mem. poisoning	b ○	●	●	●	-	✗	n.a.
ASAN [49]	mem. poisoning	- ○	●	●	●	-	✗	n.a.
LowFat [14, 16]	low fat ptrs	- ○	●	○	○	✗	✗	n.a.
EffectiveSan [15]	low fat ptrs	- ○	●	●	●	✗	✗	n.a.
SoftBound+CETS [37, 38]	ptr metadata	- ●	●	●	●	-	-	n.a.
REDFAT [17]	low fat ptrs	b ●	●	●	●	-	✗	214.5 (*)
DieHard [4]	randomization	b ●	●	●	●	-	✗	2744.9 (*)
DieHarder [42]	randomization	b ●	●	●	●	-	✗	6207.4 (*)
PACMem [30]	authentication	- ●	●	●	●	-	-	2 <sup>24</sup> (†)
HeapCheck [48]	tagged ptrs	b ●	●	●	●	-	-	2 <sup>24</sup> (†)
C <sup>3</sup> [29]	ptr encryption	b ●	●	●	●	-	-	2 <sup>24</sup> (†)
FRP	randomization	b ●	●	●	●	-	-	2 <sup>32</sup> (†)

Key: *Methodology*  
○ = none, minimal  
● = weak  
● = strong  
b = no source code needed

*Error Detection*  
○ = no support  
● = partial  
● = byte imprecise  
● = full/precise  
- = none or highly improbable  
✗ = occurs

*Entropy*  
1.0 = attack attempt  
\* = measured  
† = theoretical

## 2.1 Memory Errors

Low-level programming languages are vulnerable to *memory errors* such as *object bounds overflows* and *use-after-free*. Under the C/C++ standards [25], memory errors are *undefined behavior*, meaning that there is no prescribed semantics. For example, consider the allocation:

$$p = (T *)\text{malloc}(N * \text{sizeof}(T)); \quad (\text{ALLOC})$$

If non-NULL,  $p$  will point to the base of a new object of type  $T[N]$ . Under the C standard, the object can only be accessed via  $p$ , or a pointer *derived* from  $p$  using pointer arithmetic  $p+k$  for  $k \in 0..N-1$ . For other values of  $k$ , access is expressly undefined ([25] §6.5.6 ¶8).

Modern compilers optimize programs under the assumption that undefined behavior does not occur. If (or when) this assumption is broken, a memory error will typically access a memory location beyond the intended object, possibly including memory belonging to other objects. This behavior can be exploited. For example, an out-of-bounds write can be used to modify a function pointer stored in another object, leading to a *control-flow hijacking* attack. Similarly, an out-of-bounds read can leak sensitive information (i.e., an *information disclosure attack*), such as the infamous HeartBleed bug [18]. Use-after-free errors can be similarly exploited, by accessing a dangling pointer to a *free*'ed object, where the underlying memory has been reallocated. The importance of hardening against use-after-free should not be underestimated—around 50% of the serious vulnerabilities in Chrome are due to use-after-free [9]. History tells us not to underestimate the capabilities of potential attackers.

## 2.2 Threat Model

We model a strong-yet-realistic attacker with the following core set of *capabilities* (Cp.\*):

- Cp.Invalid**) able to construct an *out-of-bounds* pointer  $q=p+k$  or retain a *dangling* pointer;
- Cp.Deref**) able to dereference the invalid pointer  $q$ ;
- Cp.Layout**) is knowledgeable of allocated objects and their relative layout in memory; and
- Cp.Retry**) able to retry or launch multiple attacks, up to some reasonable resource bound.

The attacker uses a combination of (Cp.\*) to engineer a memory error that *accesses an object of choice*. For example, in the case of the HeartBleed bug [18], the attacker induces the program to

*construct* (Cp.Invalid) a (bad) out-of-bounds pointer of the form  $(\text{buf}+k)$  for a given buffer *buf* and value *k* beyond the length of *buf*. Note that when the attacker can control *k*, it is easy to attack defenses with low bypass resistance. The invalid pointer is then read-from (*dereferenced*, Cp.Deref), and the read value is copied into a reply message that is sent back to the attacker. The attacker also exploits basic knowledge about the *layout* of objects in memory (Cp.Layout). For example, that heap memory is *contiguous*, meaning that it is likely that other objects (and possibly sensitive information) is stored beyond the end of the buffer. Even if one attempt fails, the attacker may have the ability to *retry* attacks. Some examples include:

- (1) speculative execution attacks, like PACMan [44], allow the attacker to (speculatively) try an attack without risking program termination;
- (2) a vulnerable server may restart after a crash [1];
- (3) multiple vulnerable servers on the network;
- (4) the attacker may use common exploitation techniques, such as *heap spraying*, to increase the number of targets per attack attempt.

The ability to retry attacks (Cp.Retry) is very powerful, as it can significantly improve the probability of eventual success.

Both (Cp.Invalid), (Cp.Deref) and some combination of (Cp.Layout) or (Cp.Retry) are necessary for a successful attack. For example, if the attacker were unable to construct and dereference an invalid/bad pointer  $(\text{buf}+k)$ , then no attack would be possible. Similarly, (Cp.Layout) is also an important requirement, since the attacker must determine the correct *k* necessary for a successful attack. Sometimes the attacker may only have *partial* knowledge, e.g., that sensitive data only may be beyond the end of a buffer. However, assuming that attacks can be repeated (Cp.Retry), even partial knowledge may be sufficient. We also assume that the attacker is knowledgeable about any memory error defense used by the program. The attacker may attempt to *bypass* the defense within (Cp.\*). We refer to this as a *bypass attack*.

We note that our threat model is significantly stronger than that assumed by most existing literature. Specifically, most memory error defenses implicitly assume that an attacker can only ever attempt a single attack (i.e., no Cp.Retry), and that program termination will also terminate the attack. In contrast, our model explicitly allows multiple attack attempts, effectively allowing the attacker to “brute force” the defense.

### 2.3 Threat Mitigation

Since memory errors are a well-known problem, many existing defenses and mitigation strategies have been proposed. Some prominent examples are shown in Table 1. Here, the table summarizes the claimed<sup>1</sup> error detection capability, the underlying error detection methodology, and (where applicable) micro-benchmarks measuring attack resistance w.r.t. our threat model. Most existing defenses use (some combination of) *page-level protections*, *memory poisoning*, *low fat pointers*, *ptr metadata*, *authentication*, *encryption*, or *randomization* as the underlying methodology. We evaluate each tool under the default configuration from the publicly available repository.

Many Table 1 defenses target invalid pointer *construction* (Cp.Invalid) or *dereference* (Cp.Deref). Such defenses have well-known limitations regarding compatibility and security. For example, LowFat [14] protects pointer arithmetic (Cp.Invalid), but may also falsely flag “benign” out-of-bounds pointers<sup>2</sup> that are never dereferenced (see *false positives* in Table 1). This can result in limited compatibility with real-world code. Other defenses, such as Valgrind [40], DrMemory [7], and ASAN [49], target dereference (Cp.Deref) using *memory poisoning*, e.g., *poisoned redzones*

<sup>1</sup>Many of them assume a weaker threat model than ours (see the corresponding paper for details).

<sup>2</sup>See pointer “escapes” from [14].

inserted between objects. However, these defenses do not track pointer validity, meaning that *valid* and *invalid* access to non-poisoned memory cannot be distinguished—allowing bypass attacks (see *false negatives* in Table 1). Here, a pointer is *valid* if it points to the allocated object from which it was derived, as defined by the rules of *pointer provenance* [34]. SoftBound+CETS [37, 38] effectively implements a form of provenance tracking by explicitly attaching object metadata to each pointer, and targets (Cp.Deref). However, this solution is not binary compatible, and even the *Application Binary Interface* (ABI) must be changed to accommodate the additional metadata that must be passed for each pointer.

Fewer defenses target (Cp.Layout), and some do so only weakly. *Address Space Layout Randomization* (ASLR) randomizes the location of the heap—but provides only a very rudimentary defense between memory regions. DieHard [4] and DieHarder [42] improve on ASLR by also randomizing object locations *within* the heap. However, the attacker can still infer partial information (e.g., likely address range of allocated objects), meaning the defense is weak especially when attacks can be retried (Cp.Retry). Finally, it also is possible to allocate each object on a separate page [11, 33, 43], allowing for locations to be randomized within the limits of the virtual address space. However, placing objects on separate pages is generally impractical due to significant performance and memory overheads.

## 2.4 Problem Statement

As of writing, no memory error hardening solution has achieved ubiquitous (“always-on”) adoption. Software-only solutions tend to be too slow for practical deployment, and are also not generally binary compatible. Randomization-based and entropy-based hardware solutions show more promise, achieving both good performance and compatibility in practice.

Although some existing systems randomize pointers, they do so only weakly, and not sufficiently enough to deter our threat model. The problem is that pointers are represented as machine addresses, meaning that pointer randomization also necessitates object layout randomization. However, the degree of layout randomization is significantly limited in practice:

- (i) The x86\_64 implements a 48 bit virtual address space, meaning that the upper 16 bits of a user-mode address is sign-extended from the 17<sup>th</sup> msb (zero for user-mode);
- (ii) Due to performance and memory constraints, objects are usually allocated (near) contiguously, meaning that many objects are “close” and often share the same page(s).

To illustrate the problem, we design two proof-of-concept attacks that are allowable under our threat model: *overflow* (OF) and *underflow* (UF):

```
for (size_t i = 0; i < N; i++) attack(p+1+i); (OF)
for (size_t i = 0; i < N; i++) attack(p-1-i); (UF)
```

Here, *p* is a valid pointer to a word-sized object, and the `attack()` function attempts to access the given *invalid* (out-of-bounds) pointer. Under our threat model, the `attack()` function can effectively ignore crashes/errors, so the attack can be retried (Cp.Retry). The results are shown in Table 1 under the (*Entropy*) column, representing attack resistance. Here, (n.a.) means the defense does not attempt to resist attacks (can be interpreted as 1.0). For the others, we either measure the average<sup>3</sup> number of attempts necessary for a successful targeted attack (\*), or determine the necessary value from theory (†).<sup>4</sup> A higher value is better. The results show that, apart from this proposal, all of the Table 1 defenses are vulnerable to our threat model. For example, tools such as Valgrind and ASAN do not use any randomization, and can be trivially bypassed by choosing an

<sup>3</sup>Over 10,000 runs with a (worst-case)  $\pm 2.2\%$  margin for 95% confidence.

<sup>4</sup>We state the theoretical entropy for systems where practical experimentation is not possible.

offset  $k$  sufficient to “skip-over” poisoned redzones (Cp.Invalid). Other tools, such as DieHarder and RedFat, only use weak randomization due to limitations (i), (ii), meaning that there is a best-case  $\sim 12.6$  bits of effective security for DieHarder. For performance reasons, existing heap randomizers ultimately still allocate objects from the same memory region(s), thus tend to have limited effective entropy. Finally, existing pointer authentication, tagging, and encryption schemes use **24 bits** of entropy. Although an improvement, even this is still not sufficient against practical bypass attacks [44]. The challenge is to design a memory error defense that can resist even very strong threat models. We discuss our design below.

## 2.5 Approach

Memory errors are exploitable when the relationship between pointers is predictable. Some existing defenses attempt to counter this using randomization, but this is limited in practice leading to weak security. We believe that randomization can still be an effective defense, but the degree of randomization needs to be significantly improved. For this we design *Fully Randomized Pointers* (FRP)—a method for randomizing as many pointer bits as possible without compromising binary compatibility. We illustrate our basic approach below:

**Example 1** (Fully Randomized Pointers). To illustrate the approach, consider the allocations ( $p = \text{malloc}(10)$ ;  $q = \text{malloc}(10)$ ). Possible values for  $p$  and  $q$  could be:

<code>p=0x0000564745119010</code> ;	<code>q=0x0000564745119020</code> ;	(addresses)
<code>p=0xac8415a209566010</code> ;	<code>q=0xb5da178f9e40d020</code> ;	(fully randomized)

We assume an attacker who is aware of the value for  $p$  but not for  $q$ .<sup>5</sup> The (addresses) version is an example of an ordinary machine address returned by `glibc malloc`. Although the heap base address has been randomized using *Address Space Layout Randomization* (ASLR), the relationships between pointers are **not** random and is predictable. The attacker can easily infer the offset  $k=q-p=16$  using basic knowledge of the internals of the version of `malloc` used by the program.

In contrast, almost all of pointers bits in the (fully randomized) version have been randomized on a per-object basis—i.e., both  $p$  and  $q$  are essentially just random numbers. Given the randomized value for  $p$ , what value for  $k$  is needed to successfully overflow from  $p$  into  $q$ ? Essentially, the attacker needs to solve  $k=q-p$  where both  $k$  and  $q$  are *unknowns*, meaning that the difference  $k$  is *undetermined*. Given the high degree of randomization, brute forcing the specific  $k$  is impractical, preventing such an overflow from being constructed.  $\square$

Although full pointer randomization is our goal, the question is “how” to do so. The limitations of (i) and (ii) (Section 2.4) mean that a traditional object layout randomizer will not be sufficient and a different approach is necessary. We therefore propose a new design based on the simple premise that *pointers are not necessarily machine addresses*. Rather, we take the view that a *pointer*  $p$  is abstract entity that satisfies a set of basic properties:

- (1)  $p$  is associated with some underlying allocated object  $O$ ;
- (2)  $p$  identifies an *offset* in relation to the *base* of  $O$ ;
- (3) the standard pointer operations (arithmetic, difference, dereference, etc.) can be applied to  $p$ .

A machine address is one possible pointer encoding, but other encodings are possible. Our hypothesis is that most existing binary code is agnostic about the underlying pointer encoding, just provided these basic operations have the expected semantics. Thus, we can replace the pointer encoding to introduce stronger memory safety, while retaining compatibility without recompilation.

<sup>5</sup>For example, with HeartBleed, the attacker knows  $p=\text{buf}$ , and must use  $p$  to construct the address  $q$  of the target object.

We design a **new** pointer encoding based on abstract  $\langle id, offset \rangle$  pairs, where  $id$  is a unique identifier associated to a specific object  $O$ , and  $offset$  is the explicit byte offset within  $O$ . We also instantiate the standard pointer operations (arithmetic, dereference, etc.), and show how this is sufficient for binary compatibility. Importantly, the abstract encoding is:

- (a) *decoupled* from the underlying object memory address; and
- (b) both  $id$  and  $offset$  are *strongly randomized*—allowing for the relationship between pointers to be obfuscated and protecting against bypass attacks

By “decoupling” the pointer encoding from the underlying memory address, the limitations of (i) and (ii) are also no longer of concern—a key difference between our approach and existing heap randomizers (DieHard(er) [4, 42]). Furthermore, even (partial) knowledge of object layout can no longer be exploited (Cp.Layout), and the number of retries necessary for a brute-force style attack will exceed the attacker’s resources (Cp.Retry).

The final challenge is performance. In general, software implementations of memory safety tools have high overheads, and are far too slow for widespread practical use, so we focus on a hardware implementation. Since pointers are not represented as machine addresses, dynamic pointer decoding is necessary. Furthermore, the mapping between pointers and addresses can change at any time as objects are `malloc`’ed and `free`’ed. For performance we use an old idea: a *cache* that can quickly retrieve the most frequently used pointer decodings. We will discuss a hardware implementation (GREENFAT) of these ideas.

## 2.6 Detailed Example

We implement *Fully Randomized Pointers* (FRP) by intercepting memory dereference operations and dynamically decoding pointer values. A detailed example of on-the-fly pointer decoding is illustrated in Figure 2. Here, we consider an example instruction with a (randomized) encoded pointer stored in `%rax`. The *Load Effective Address* (LEA) operation evaluates the memory operand `0x2(%rax,%rbx,2)`, which corresponds to the pointer arithmetic ( $0x2 + \%rax + 2 \times \%rbx$ ). This operation is executed as normal under the standard x86\_64 semantics, but this time generates a derived encoded pointer rather than an address. The dynamic pointer decoder is inserted in between the (LEA) and dereference (`*`) operations. Here, the encoded access pointer is *mapped* (MAP) into an (encoded) base pointer ( $q$  from Example 1), and a (decoded) base memory address. The difference ( $-$ ) between the (encoded) access pointer and base pointers is then added ( $+$ ) to the (decoded) base to yield the final (decoded) memory address. The final address is dereferenced ( $*$ ) as per normal, before being discarded. As such, only fully randomized (encoded) pointers are visible to the program, and these have no observable relation to the underlying memory address.

In addition to pointer decoding, our approach also checks for memory safety. For *use-after-free* errors, our approach ensures that encoded pointers are never reused before resource ( $id$ ) exhaustion. This means that dangling pointer access will be detected by a missing entry in the (MAP)—even when the underlying memory has been reallocated. To check for *object-bounds* errors, (MAP) also

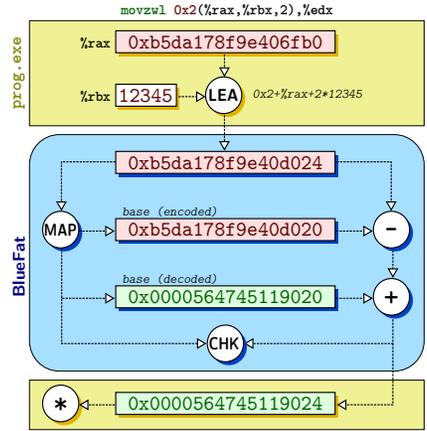


Fig. 2. Example pointer decoding.

stores object *metadata* (size+base address) which is checked before access (CHK):

$$[base, base+size) \cap [lb, ub) = [lb, ub) \quad (\text{CHK})$$

Here, *base* is the object base address, *size* is the object size, and  $[lb, ub)$  are the bounds of the memory access. Assuming the object in Figure 2 is of size 10 bytes (Example 1), the resulting memory access is within bounds and thus is allowable. Otherwise, if the address is out-of-bounds, our approach will detect this case and take protective action before the dereference operation (\*).

Finally, we must also consider the tricky case where the base address `%rax` is itself invalid. For example, given the valid pointer *p* from Example 1, an attacker could (in theory) pass the *invalid* (out-of-bounds) pointer  $p' = p + k$  into `%rax`. This is possible since pointer arithmetic is never checked to avoid *false positives* at the binary level. If the invalid  $p'$  were to have the same value as a valid pointer *q*, then the invalid access will not be detected by either the (MAP) nor (CHK) checks—effectively *bypassing* the defense. Indeed, other memory safety tools, such as Valgrind [40] and DrMemory [7], are vulnerable to precisely this kind of bypass attack. Essentially, the attacker needs to find the correct value for *k*, which is easy to do with default memory allocators that allocate objects within the same memory region. In contrast, FRP protects this case using a combination of pointer decoupling and randomization. Namely, under our running example, the attacker must generate  $p'$  from *p* by finding the very specific value  $k = 672727314255736848$  out of  $\sim 2^{52}$  possibilities. The attacker cannot exploit basic memory layout knowledge (Cp.Layout) since pointer values are decoupled. Furthermore, the high degree of randomization means that the number of retries (Cp.Retry) necessary will easily exceed any practical attacker resource.

The problem of invalid pointers is closely related to the notion of *pointer provenance* [34], which refers to the relationship between pointer values and the underlying object that the object pointer references. It is possible that two pointers have the same value but have different provenance (i.e., reference different objects), as is the case with the (invalid)  $p'$  and valid *q* in the example above. For compatibility, tools based on memory poisoning (ASAN, Valgrind, etc.) do not track pointer provenance and are therefore vulnerable to bypass attacks using invalid pointers. Other memory safety solutions, such as SoftBound [37] explicitly track object metadata associated with each pointer, and therefore do effectively track pointer provenance. However, the explicit metadata tracking is not binary compatible, and must be re-compiled from source code. FRP is effectively a form of *implicit* provenance tracking under a probabilistic guarantee, since different provenances correspond to very different (randomized) pointer values, and cannot easily be confused. Unlike explicit tracking, FRP does not require special handling of pointer arithmetic, and thus preserves the ABI and achieves strong binary compatibility. Since FRP values are ordinary machine words, even advanced pointer idioms, such as XOR-linked-lists, are fully supported. Such idioms tend to break explicit metadata tracking.

Next, we shall detail our design in two main parts: pointer decoupling, and randomization.

### 3 DECOUPLED POINTERS

In this section, we propose a *decoupled* pointer encoding that is binary compatible and independent of the underlying memory layout, defending against (Cp.Layout). Our approach is to first define the necessary pointer requirements, then to design a decoupled encoding accordingly.

#### 3.1 Pointer Requirements

For compatibility, the pointer encoding must satisfy a set of basic requirements assumed by most binary code. Here, each allocated object *O* is uniquely identified by a *base pointer* *p* returned by the underlying allocator. The main pointer requirements are:

- Rq.Arith**  $p_i = p + i$  points to the  $i^{\text{th}}$  byte in  $O$ ; and  $p_j = p_i + k$  (and  $k = p_j - p_i$ ) for  $k = (j - i)$ .  
 Pointer comparison is defined in the obvious way.
- Rq.Word**  $p_i$  is representable as a (64 bit) machine word.
- Rq.Deref**  $p_i$  can be dereferenced to access  $n$  consecutive  $O$  bytes pointed to by  $p_i \cdot p_{i+n-1}$ .
- Rq.Align**  $(p \ \& \ 0\text{xFFF})$  must equal the alignment of  $O$ .

A more detailed treatment of pointer requirements can be found in [34]. The (Rq.Align) requirement is often necessary for (binary) compatibility, where the binary determines object alignment by inspecting the pointer value.<sup>6</sup> Most real-world programs also relax some of the requirements. For example, many C/C++ programs (and most binaries) relax (Rq.Arith), allowing for arbitrary  $k$  values outside the bounds of  $O$ . In the case of C/C++, this is technically *undefined behavior* ([25] §6.5.6 ¶8), but is nevertheless relied upon by many programs [17]. Furthermore, many binaries relax (Rq.Deref) for out-of-bounds *reads*, provided that (1) the memory access does not cause a page fault (SIGSEGV), and (2) any out-of-bounds data is discarded by subsequent execution. For performance reasons, many `glibc` string handling functions (`strlen`, `strcpy`, etc.) use SIMD instructions that may read (then discard) out-of-bounds data. We claim that any pointer encoding satisfying the relaxed (Rq.\*) will be compatible with most existing binary code.

### 3.2 Decoupled Pointer Design

We design a basic decoupled pointer encoding that satisfies the relaxed (Rq.\*). This subsection is necessarily low-level, since the bit-level encoding directly relates to compatibility and security.

*Pointer Arithmetic.* Our underlying approach is to refactor pointers into explicit  $\langle id, offset \rangle$  pairs:

- $id$  is an *object identifier* that uniquely identifies some allocated object  $O$ ; and
- $offset$  represents the byte offset from the base of  $O$ .

With this abstract encoding, a memory allocator returns a pointer of the form  $\langle id, zero \rangle$ , where  $id$  is a freshly minted (never-used-before) identifier, and  $zero$  represents an offset of 0 bytes. Here,  $zero$  can be any arbitrary zero-point value, but for now we shall use the obvious value  $zero = 0$ .

Under the abstract encoding, the  $id$  does not depend on the machine address—rather, objects are associated with unique identifiers that are never reused, even when the underlying memory is freed and reallocated. For this, identifiers could even be assigned sequentially (#1, #2, #3, ...), meaning that a dangling abstract pointer can never be confused with a valid pointer to a reallocated object. We also use the following definition of pointer arithmetic:

$$\langle id, offset \rangle + k = \langle id, offset + k \rangle \quad (\text{ARITH})$$

That is, pointer arithmetic only affects the *offset*, meaning that an out-of-bounds abstract pointer can never be confused with a valid pointer to another object. The object identifier ( $id$ ) is idempotent with respect to reallocation and pointer arithmetic, resolving (in principle) all potential ambiguity between invalid and valid pointers.

*Flattening.* To satisfy (Rq.Word), we *flatten* the abstract encoding into 64 bit integers:

```
struct { int64_t id:40; int64_t offset:24; };
```

Here *offset* bitfield occupies the  $m = \log_2(M)$  least significant bits (lsb), and the  $id$  occupies the remaining  $64 - m$  most significant bits (msb), where  $M$  based on the default `glibc malloc` threshold<sup>7</sup>  $M = \text{MMAP\_THRESHOLD\_MAX} = 32\text{MB}$ . Note that the *offset* is stored within the lsb, meaning that integer encoding is consistent with (ARITH) for small values of  $k$ , satisfying (Rq.Arith). Finally, to satisfy

<sup>6</sup>Most programs are compatible with 16-byte alignment (0xF). However, some programs are sensitive to page alignment (0xFFF), specifically when interacting with system calls (e.g., `mprotect`) where page alignment is required.

<sup>7</sup>See the `mallocpt` manpage.

requirement (Rq.Align), we use  $zero=(addr \& 0xFFF)$ , thereby preserving the page offset (12 lsb) of the machine address ( $addr$ ).

*Pointer Dereference.* The final requirement is pointer dereference (Rq.Deref). To address this, our approach is to intercept all memory accesses and to decode pointers “on-the-fly”, as illustrated in Figure 2. Essentially, the pointer dereference ( $*p$ ) operation is replaced with  $*decode(p)$ , where  $decode$  maps encoded pointers to the corresponding machine address, defined as follows:

$$decode(p) = \begin{cases} \mathcal{M}[p.id].base + p.offset & isEncoded(p) \\ p & otherwise \end{cases}$$

Here,  $isEncoded$  determines whether a pointer is encoded or not. For the  $x86\_64$ , we can assume that a non-zero value for any of the 16 most significant bits will indicate an encoded pointer. The mapping  $\mathcal{M}$  is an *associative map* that maps identifiers of allocated objects to information about the corresponding object (also see MAP from Figure 2). Here,  $(\mathcal{M}[id].base)$  is the *base machine address* of the object with  $id$ . Initially the map is empty ( $\mathcal{M}=\emptyset$ ) when the program starts. As execution proceeds,  $\mathcal{M}$  is updated each time an object is (de)allocated, using wrappers over standard memory allocation functions.

The  $\mathcal{M}$  map can be implemented using an associative map data structure. However, querying an associative map is a relatively slow operation, especially since  $(\mathcal{M})$  must be consulted for each memory access. In hardware, we can optimize memory access wrapping the  $\mathcal{M}$  map with a *cache* ( $\mathcal{C}$ ) to store the most frequently accessed objects. The impact of  $\mathcal{C}$  can be significant when the *miss ratio* is low, and we shall evaluate the impact of this design in Section 5.

### 3.3 Discussion

Decoupled pointers satisfy the relaxed requirements (Rq.\*), and are a practical encoding that is compatible with existing binary code (without recompilation) assuming that pointers can be decoded on-the-fly. Although the decoupled pointer encoding is compatible, it by itself only provides weak protection against bypass attack. For example, (1) the *offset* bitfield may *overflow* into the *id*, meaning that  $q=p+k$  is still has a solution, and (2) the *id* bitfield has a limited size (40 bits), meaning that identifier values will need to be reused eventually (after  $\sim 10^{12}$  allocations). We address security using full randomization in Section 4.

Decoupled pointers also assume that all pointer dereference operations can be intercepted for pointer decoding. For a software implementation, this can be achieved using a *Dynamic Binary Instrumentation* (DBI) framework—similar to the approach used by other DBI-based memory error checking tools (e.g., Valgrind [40] and DrMemory [7]). The main downside of DBI is the performance overhead, meaning that a software-only implementation of decoupled pointers would be too slow for most practical use cases. However, the approach is feasible as a hardware solution, where pointer decoding can be integrated into the CPU load-store pathway with little performance impact. We shall present a hardware implementation in Section 5.

## 4 FULLY RANDOMIZED POINTERS

We use the decoupled pointer encoding from Section 3 as the basis for *Fully Randomized Pointers* (FRP). For binary compatibility, all pointers must be represented as 64 bit integers (Rq.Word) meaning that the object identifier bitfield (*id*) is vulnerable to modification, and this is difficult to prevent directly. Even if all pointer arithmetic were to be fully instrumented (e.g., to detect bitfield overflows), this would break compatibility with binary code that creates deliberate out-of-bounds pointers (relaxed Rq.Arith). Instead, we use an indirect mitigation in the form of full randomization.

#### 4.1 Decoupled Pointer Randomization

The idea is to obfuscate that relationship between pointers to different objects, meaning that the necessary value for  $k$  to construct an out-of-bounds pointer (`Cp.Invalid`) becomes undetermined. Similar protection is afforded to dangling pointers, thereby hardening against bypass attacks in general. Unlike existing randomization-based defenses, such as `DieHard(er)` [4, 42], our baseline pointer encoding is decoupled from the underlying memory address, meaning that we randomize pointer bits without affecting the layout of objects within memory. Decoupled pointer randomization is relatively straightforward: the *identifier* ( $id$ ) and *offset* bitfields are randomized independently, as discussed below.

*Identifier randomization.* Identifiers can be generated at random with a collision-check:

$$\text{generateId}() = \begin{cases} id & id \leftarrow \text{getId}(), id \notin \text{dom } \mathcal{M} \\ \text{generateId}() & \text{otherwise} \end{cases}$$

Here,  $\text{getId}()$  is a *Cryptographically Secure Pseudo Random Number Generator* (CSPRNG) source. Since different objects will be assigned very different (random) ids, there is no predictable relationship between their respective pointers, defending against (`Cp.Invalid`).

*Offset randomization.* It is also possible to randomize the *offset*. This exploits the observation that the zero-point (*zero*) can be any arbitrary value subject to the following constraints:

- (1) ( $zero + size$ ) cannot overflow the offset bitfield, and
- (2) ( $zero \& 0xFFF$ ) must match the decoded machine base pointer in order to satisfy (`Rq.Align`).

Any random (CSPRNG-sourced) value satisfying (1) and (2) can be used for the *zero* value. Offset randomization further obfuscates the relationship between pointers to different objects, further hardening against (`Cp.Invalid`).

*Additional optional randomization.* It is also possible to (optionally) randomize the page offset as an extension in order to further harden the defense. For this, we can choose a memory allocator that randomizes object locations *within each page*. This also assumes that an attacker is unable to predict page offset randomization (`Cp.Layout`), which may be the case in many common attack scenarios.

Normally, heap objects must be “*suitably aligned for any built-in type*”<sup>8</sup>. In practice for the `x86_64`, this means that heap objects are aligned to 16-byte boundaries. As an extension, relaxing this assumption allows the remaining pointer bits to be randomized—further maximizing security. However, some programs rely on the default `malloc` alignment, so this extension may break compatibility, so is disabled by default.

*Sensitive data.* An attacker may also attempt to bypass the defense by reading sensitive data, such as the mapping  $\mathcal{M}$ , CSPRNG-parameters, or the heap itself (via a machine address). For our analysis, we assume that sensitive data is stored in a dedicated memory region that is not accessible by the program, i.e., *not* addressable. For a software implementation, this can be achieved with additional instrumented checks. For hardware, this can be achieved by storing sensitive information in an inaccessible (e.g., non-addressable) memory region.

Note that the heap itself is considered sensitive data. This means that heap objects are only accessible via FRPs and not regular addresses. This assumption is critical for our security analysis.

<sup>8</sup>See the `malloc` man page.

## 4.2 Security Analysis

We model two single-word objects pointed to by encoded pointers  $p, q$ . We assume that the attacker knows  $p$  and  $q$  is the intended target. Due to the randomization, the attacker knows the value of  $p$  but not  $q$ , which can be any value in the FRP space. The attacker can construct (Cp.Invalid) and dereference (Cp.Deref) invalid pointers. Furthermore, the attacker is able to retry failed attacks (Cp.Retry). The attacker is adversarial, meaning that they can attempt any attack allowable under the threat model.

We can quantify the probability of an isolated attack attempts succeeding as follows:

$$(2^{|id|} - 1 - 2^{|id|-16}) \times 2^{|offset|} \times 2^{|page|} \times 2^{|align|} \quad (\text{SECURITY})$$

Here,  $|id|$  is the number of randomized bits in the  $id$ ,  $|offset|$  is the number randomized bits in the  $offset$ ,  $|page|$  is the number of randomized bits in the page offset (optional), and  $|align|$  is the remaining number randomized bits that is otherwise reserved for `malloc` alignment (optional). Here (SECURITY) accounts for  $2^{|id|}$  potential randomized identifiers, less one already consumed by  $p$ , less  $(2^{|id|-16})$  identifiers with zero 16 msb. For our baseline, we assume  $|id|=40$ ,  $|offset|=12$ , and  $|page|=|align|=0$  which evaluates to **52 bits**—significantly stronger than the current state-of-the-art [29, 30, 48]. This analysis conservatively assumes the attacker is aware of the memory layout (Cp.Layout) and can therefore guess the lower 12 bits of the target  $q$ . By relaxing these assumptions, we have that  $|page|=8$  and  $|align|=4$ , meaning that the effective entropy increases to **60 bits** and **64 bits** respectively. We now consider some possible attack scenarios:

*Network server reset.* The attacker attempts to dereference an invalid pointer, hoping to hit the target  $q$ . For an incorrect guess, the program terminates, but the attacker can retry under (Cp.Retry). Under the baseline entropy of 52 bits, the attacker will be required to make an average of  $2^{52-1}$  guesses before a successful attack. Assuming the attacker targets a network server that resets after a crash, the network bandwidth for the TCP handshakes ( $3 \times 40$  bytes) alone would exceed **240 petabytes**. As such, this is not a practical attack.

*Speculative execution.* Here the attacker dereferences invalid pointers *speculatively*, and uses side channels (e.g., timing) to determine whether the guess was valid or not. The advantage of this approach is that it does not terminate the program on an unsuccessful attempt. The disadvantage is that it requires a more complicated set-up, and may require sampling of noisy side channels.

For our analysis, we use the PACMan attack as a guide, which is a practical speculative execution attack against ARM *Pointer Authentication Codes* (PAC) [44]. We use the estimate of 2.69 milliseconds per attack attempt ([44] § 8.2), which can exhaust all 16 bit PAC values in 2.94 minutes, and all 24 bit values in 12.54 hours. At the same rate, exhausting all 52 bit FRP values would take over **192077 years**. Again, this is not a practical attack.

*Heap spray.* In some cases, the attacker could induce the program to create multiple  $n$  targets (heap spray), rather than assuming a single target ( $n=1$ ). However, we can consider this to be another form of (Cp.Retry), where  $n$  is the number of attack attempts. The disadvantage of this approach is that each allocated target consumes memory, which imposes a practical bound on the size of  $n$ . Furthermore, memory size is orders of magnitude smaller than the size of the FRP space, meaning that heap sprays have negligible impact on security. For example, suppose the attacker allocated  $2^{32}$  targets, which by itself would consume considerable memory. However, the effective security of FRP would be essentially unchanged:  $2^{52} - 2^{32} \approx 2^{52}$ .

The attacker may also attempt to allocate objects to reduce the size of the (unallocated) FRP space, making future allocations more predictable. However, this runs into the same problem: the FRP is so large that it will not be meaningfully affected by such attacks, using the same equation

as above. Furthermore, such an attack would require the attacker to somehow track the values of previously allocated pointers, which may not be feasible under our threat model.

*Discussion.* The large entropy of FRP protects against even “brute force”-style attacks, even under conservative assumptions. The large entropy may also help protect against future hypothetical attacks (e.g., targeting a flawed implementation) that reduces the effective entropy, since FRP provides a maximized entropy buffer.

## 5 EVALUATION

We have implemented a software implementation (BLUEFAT) of *Fully Randomized Pointers* (FRP) on top of the *Pin Dynamic Binary Instrumentation* (DBI) framework [32] and a hardware simulation (GREENFAT) on top of gem5 [5]. The software implementation primarily aims to comprehensively test the security and compatibility of FRP on real-world x86\_64 binaries without recompilation. The hardware simulation aims to evaluate the performance potential of FRP under the assumption that the cache (*C*) is implemented natively in hardware.

In both cases, the FRP wraps the standard `glibc` memory allocation functions with full pointer randomization, and intercepts all memory dereference to dynamically decode pointers into the corresponding memory address. The implementation also checks for memory errors, and will abort the program on use-after-free or out-of-bounds write. For reads, the implementation will zero any out-of-bound byte (similar to *failure-oblivious computing* [45]), retaining compatibility with the relaxed (`Rq.Deref`) while also preventing exploitation in the form of information disclosure. In terms of look-and-feel, the software implementation (BLUEFAT) is similar to other DBI-based memory error tools, such as Valgrind [40] and DrMemory [7]. However, unlike these related tools, the FRP methodology is primarily designed to harden against bypass attacks rather than for bug detection.

All experiments are run on an Intel Xeon Gold 6242R CPU (clocked at 3.10GHz) with 376GB of RAM. Each benchmark binary is obtained by compiling using `gcc/g++/gfortran` compiler version 9.4.0 under (`-O2`). The system runs on Ubuntu 20.04 (LTS). We evaluate BLUEFAT for security/compatibility and GREENFAT for performance.

### 5.1 Security

FRP is primarily optimized for security and resistance to bypass attacks. To evaluate the effectiveness of FRP, we evaluate BLUEFAT against several prominent memory error defenses from Table 1. For benchmarks we use (in increasing order of difficulty):

- (1) Recent *Common Vulnerabilities and Exposures* (CVEs);
- (2) REDFAT benchmarks (from [17]) including *non-incremental* [17] CVEs and 480 Juliet [41] test cases;
- (3) A *strong* adversarial attacker model capable of brute-force bypass attacks (see Section 2.2). This includes ported REDFAT CVEs and the Table 1 microbenchmarks.

We caution against drawing conclusions from the *quantity* of passing/failing tests. For example, the REDFAT benchmarks from [17] (which are used “as-is”) are tuned against Valgrind. Rather, *any* failing test means that the tool is vulnerable to bypass attacks (see the *Bypass safe?* column). We also note that, unlike the other tools, DieHard(er) implements error detection *mitigation* rather than detection. This is represented by a dash (-) and should not be interpreted as a negative outcome. We evaluate all tools from Table 1 for which implementations are publicly available and can run under our test set-up.

The results are shown in Table 2. Here, we see that most existing defenses are able to detect “ordinary” CVEs (*Recent*) during normal execution—i.e., without an attacker that is actively attempting

Table 2. Security evaluation of defense techniques.

Defense	Recent				REDFAT Bench.				Strong			Bypass safe?				
	CVEs				CVEs				Jul.	CVEs			Micro			
	cve-2023-29584	cve-2023-25221	cve-2023-27249	cve-2023-25222	cve-2007-3476	cve-2012-4295	cve-2016-1903	cve-2016-2335	cve-122-heap	cve-2007-3476*	cve-2012-4295*	cve-2016-1903*	cve-2016-2335*	OF+UF	UAF	
ASLR+glibc	X	X	X	X	X	X	X	X	0	X	X	X	X	X	X	X
ASAN [49]	✓	✓	✓	✓	✓	✓	✓	✓	210	X	X	X	X	X	X	X
LowFat [14, 16]	X	X	X	✓	✓	X	X	X	428	X	X	X	X	X	X	X
EffectiveSan [15]	✓	✓	X	✓	✓	✓	✓	✓	480	X	X	X	X	X	X	X
efence [43]	X	✓	X	✓	✓	✓	✓	✓	480	X	X	X	X	X	X	X
DieHard [4]	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X
DieHarder [42]	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X
Valgrind [4]	X	✓	✓	✓	X	X	X	X	0	X	X	X	X	X	X	X
DrMemory [7]	X	✓	✓	✓	X	✓	✓	X	476	X	X	X	X	X	X	X
REDFAT [17]	✓	X	X	✓	✓	✓	✓	✓	480	X	X	X	X	X	X	X
FRP (this work)	✓	✓	✓	✓	✓	✓	✓	✓	480	✓	✓	✓	✓	✓	✓	✓

(✓) = error detected, (X) = error not detected

(-) = *indeterminate* since the benchmark does not model a target object.

(Jul.) = Juliet benchmarks (480 tests); (\*) = ported to strong attacker

to bypass the defense. Such tools are useful for debugging or at least some modest attack surface reduction. The REDFAT benchmarks use *non-incremental* overflows which allow for the construction of pointers with attacker-controlled offsets ( $p+k$ ). This leads to mixed results on some defenses—the value chosen for  $k$  may be sufficient to bypass common detection methods, e.g. memory poisoning. Finally, we consider our “strong” threat model (*Strong*). For this, we port the REDFAT CVEs to also include a proactive attacker with the (Cp.\*) capabilities, including the ability to retry (brute force) failed attacks. We also test the micro benchmarks from Section 2.4, as well as an additional *use-after-free* (UAF) benchmark:

```
free(p); for (size_t i = 0; i < N; i++) { malloc(n); attack(p); } (UAF)
```

For these tests, we use a maximum of 10,000 attempts. Only BLUEFAT offers any real resistance. Since pointers have been randomized, the number of attempts necessary for a successful attack is  $\sim 2^{51}$ , providing strong security even against the capable attacker. Table 2 also highlights the problem of incomplete error coverage due to uninstrumented code (e.g., shared libraries), as illustrated by CVE-2023-27249, which is missed by many tools. Such problems are avoided by transparently checking *all* memory access, which is possible with a hardware implementation (GREENFAT).

Our results also reinforce the earlier results from Table 1. Namely, that existing defenses are effective against weak or non-existent attacker models (*Recent*), have mixed results against less-weak models, and have essentially no resistance against strong attacker models (*Strong*). Given that the strong attacker model is not unrealistic, this highlights the need for stronger memory error defenses such as FRP.

## 5.2 Compatibility

To test compatibility, we evaluate BLUEFAT against the full SPEC2017 benchmark suite [8] under the *ref* workload. There are 19 benchmarks in total, with 12 benchmarks using C, 5 using C++, and 8 benchmarks using Fortran (non-exclusively). The results are shown in Figure 3, and represent the relative slowdown using Valgrind’s Memcheck as a baseline. To make the comparison fairer, we

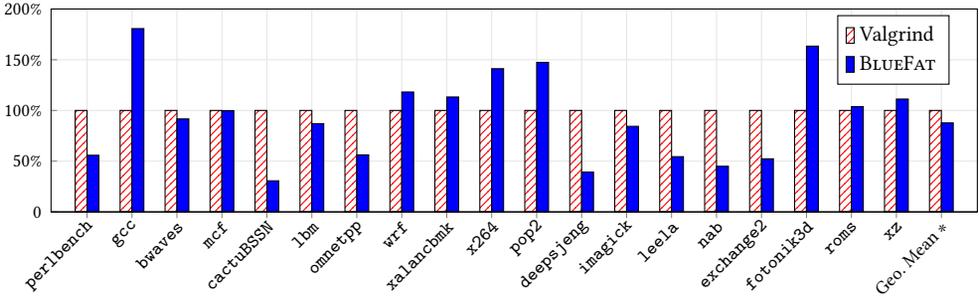


Fig. 3. Normalized SPEC timings for Valgrind and BLUEFAT. For (\*) we exclude `cam4`, which runs under BLUEFAT but not Valgrind. The results demonstrate perfect compatibility for BLUEFAT and FRP—despite using a radically different pointer encoding.

disable leak checking (`-leak-check=no`) and undefined value errors (`-undef-value-errors=no`), which speeds up Valgrind somewhat. We compare against BLUEFAT with a software cache size of 64KB (4096 entries).

We compare against Valgrind as a *Dynamic Binary Instrumentation* (DBI)-based memory error detection tool that is binary compatible without recompilation. We caution that compatibility (and not performance) is the primary goal of these experiments—i.e., can FRP (as implemented by BLUEFAT) run the full SPEC2017 benchmark suite without error? It is generally accepted that DBI is too slow for most hardening applications, and that only a hardware-based implementation can deliver performance overheads acceptable for general deployment. Our main performance result is based on a hardware simulation (GREENFAT), and will be presented in Section 5.3.

Binary compatibility is essential for a usable tool. Despite using a radically different pointer encoding (i.e., FRP replacing addresses with random numbers), BLUEFAT can successfully run the entire SPEC benchmark suite (Figure 3). Furthermore, each binary runs without any special (re)compilation or instrumentation, i.e., the binary itself is unaware of the hardening solution. This validates the key premise of our design, namely: alternative pointer encodings are indeed practical provided that the minimum set of pointer requirements (Rq.\*) have been satisfied. The results demonstrate that FRP can be applied to any binary, including legacy binaries and libraries, without the need to rebuild the entire software tool chain. We also tested BLUEFAT on a wide plethora of off-the-shelf binaries, including ([12] Table II *Real-World*), ([13] Table 1), and many other programs/libraries. We find that BLUEFAT is highly compatible with existing binaries—further validating our approach. However, there are some exceptions, which will be discussed in Section 5.4.

Other memory safety tools have mixed results regarding compatibility, despite using the default pointer encoding. Valgrind can run most benchmarks with the exception of `cam4` (which failed due to a segmentation fault under our tests). We also tested DrMemory [7], with `deepsjeng` reporting a heap allocation failure, but with the other benchmarks running, albeit much slower than Valgrind. While other DBI-based memory error detection tools can also achieve reasonable compatibility, only FRP achieves the strong bypass resistance and security guarantees necessary for hardening applications.

### 5.3 Performance

In this section we demonstrate a proof-of-concept hardware-accelerated version of FRP (GREENFAT). Our goal with GREENFAT is to propose a hardware-software co-designed solution that significantly reduces the overheads seen when using FRP. It does this by storing a mapping from the FRP value to the virtual address and bounds metadata in a small, fast-to-access cache. This accelerates the

translation of an FRP and allows for the hardware to quickly check recently used FRP values for the translations in the small cache. In the relatively rare case of a cache miss, we access the translation data from DRAM and cache it for future accesses.

In the rest of this section, we describe the methodology used for application representative generation, detailed simulator setup, and provide an analysis of the hardware simulation results.

*5.3.1 Methodology.* We model our FRP implementation, GREENFAT, using the gem5 [5, 31] simulator. The gem5 simulator is a popular cycle-level processor simulation methodology. We model a modern processor system with a full cache hierarchy, and a detailed out-of-order CPU execution model (See Table 3 for details) in Syscall Emulation mode. We then augment this system with the GREENFAT cache to build a fast-to-access memory that tracks recent object-to-bounds translations, minimizing overheads and allowing for fast lookups of recently-used FRP mappings.

The GREENFAT cache is modeled with {128, 512, 1024, 4096} entries, and each entry stores the virtual address bounds information for each object (assumes 16 B of space per entry, same as BLUEFAT). The total capacity for these caches are {2 KiB, 8 KiB, 16 KiB, 64 KiB} respectively. Entries are added to the FRP mappings through our memory allocation wrappers (`malloc()`, `free()`, etc.) that do not require recompilation of the binaries under test (the use of `LD_PRELOAD` to insert `malloc` wrappers is sufficient for these workloads). The cache itself is connected directly to the memory controller to avoid LLC cache pollution. All misses will access DRAM in a deterministic, non-cached way to prevent potential side-channel attacks from partially decoding pieces of the FRP information.

The evaluation is done using the working subset of C/C++ applications<sup>9</sup> from the SPEC CPU2006 benchmark suite<sup>10</sup> built using LLVM 10.0 compiler with a RISC-V backend. As running full applications in detail can take months to simulate in gem5 [47], we therefore use application representatives that accurately represent the original workload behavior [21]. To generate these representatives, we use functional simulation of gem5 (AtomicCPU mode) which allows us to collect the Basic Block Vectors (BBVs) required to cluster the regions to generate the 1 billion instruction SimPoint [21] representatives. The full reference input sets were used (largest input sets) running both the (i) baseline gem5, and the (ii) GREENFAT configuration. We collect performance and cache statistics, and detail those results later in this section. All object statistics were generated using the 1 B representative region execution.

*5.3.2 GREENFAT Implementation Details.* In this section, we will present an overview of the steps taken by the hardware-software co-designed system to implement FRPs. First, (1) upon heap memory allocation, we allocate a mapping from a newly minted FRP to a virtual address region. This is done at the time of the `malloc()` function call. Next, (2) on access by a standard load or store instruction, the hardware checks for the presence of an FRP, and, when identified, (2a) looks up the FRP in the GREENFAT cache to determine the object base and bounds. On a hit, the access continues as normal, but on a miss (2b), the GREENFAT cache reads the object map in DRAM, and stores the entry in the cache for later use. The hardware next (3) checks for validity of the address, and if (3a) valid, proceeds to access the L1-D cache using the virtual address from the FRP and the offset supplied from software. Note that data from the GREENFAT mapping needs to be checked (either via the cache lookup, or from DRAM) before initiating an L1-D cache lookup via the translated virtual address. If invalid (3b), the hardware will raise a memory access exception (similar to an invalid memory access), and the OS will subsequently raise a SIGSEGV to the application.

<sup>9</sup>For the setup used to build representatives, the RISC-V compiler used (LLVM 10.0) does not support Fortran applications, and the gem5 version used has many unimplemented syscalls, limiting application support similar to previous work [20].

<sup>10</sup>CPU2006 is compatible with our gem5 setup.

Table 3. System configuration used in gem5 simulation. Acronyms used below include Out-of-Order CPU (OoO), Load Queue (LQ), Store Queue (SQ) and Reorder Buffer (ROB), Least-Recently Used (LRU).

Component	Description
CPU	OoO @ 1 GHz, 32 LQ, 32 SQ, and 192 ROB entries
GREENFAT Cache	8-way LRU, 2 cyc hit lat, 300 cyc miss lat, 16 B line size
L1I/D Cache	32 KiB, 2-way LRU, 2 cyc lat, 64 B line size
L2 Cache	256 KiB, 8-way LRU, 20 cyc latency, 64 B line size
DRAM	tCL 13.75 ns, tRCD 13.75 ns, tRP 13.75 ns (approximately 80 cycles)

Table 4. Benchmark slowdown (%) and cache miss rates (%) for gem5 simulations of our GREENFAT implementation across different cache sizes (4096, 1024, 512 and 128 entries of a 8-way associative cache). Slowdown is calculated by comparing the benchmark simulated runtime under GREENFAT with baseline gem5. In addition, object data for the representative regions are listed for comparison. Entries with a hyphen (-) indicate a cache miss rate of less than 0.01%. Count is the number of unique objects accessed in the representative region. The  $\mu$  symbol represents the mean object size (in KiB), and  $|A|$  is the number of accesses, in millions.

Benchmark	Slowdown (%)				Cache Miss Rate (%)				Object Data		
	4096	1024	512	128	4096	1024	512	128	Count	$\mu$ (KiB)	$ A $ (M)
401.bzip2	2.54	2.54	2.54	2.54	-	-	-	-	7	7,329.8	171.4
403.gcc	2.60	3.13	3.52	12.61	0.03	0.04	0.06	0.46	8,901	2.5	230.9
429.mcf	4.98	4.98	4.98	4.98	-	-	-	-	3	190,698.8	1043.7
433.milc	1.94	1.94	1.94	1.94	-	-	-	-	40	11,798.5	357.1
445.gobmk	0.22	0.22	0.22	0.22	-	-	-	-	8	1,487.6	18.5
462.libquantum	0.74	0.74	0.74	0.74	-	-	-	-	1	8,193.0	253.0
464.h264ref	0.42	0.77	1.69	5.36	0.00	0.01	0.05	0.13	1,164	1.8	292.7
470.lbm	3.98	3.98	3.98	3.98	-	-	-	-	2	52,344.8	311.2
473.astar	59.57	63.10	66.02	69.30	18.89	19.93	21.31	22.83	192,298	0.5	63.0
483.xalancbmk	18.99	52.01	121.23	242.97	0.59	1.43	2.94	5.12	66,444	0.6	231.0
GEOMEAN	8.46	11.49	16.11	23.02	-	-	-	-	-	-	-

5.3.3 *Analysis.* We assess GREENFAT cache statistics and the impact on performance over four different cache-sizes — 2 KiB, 8 KiB, 16 KiB, and 64 KiB corresponding to 128 entries, 512 entries, 1024 and 4096 entries, respectively. Table 4 displays the GREENFAT cache miss rates and performance slowdown of GREENFAT compared to the baseline gem5 implementation. As GREENFAT stores mappings in the GREENFAT cache on a per-object basis, the miss rate remains low (<0.01%) for six benchmarks (401.bzip2, 433.milc, 429.mcf, 445.gobmk, 462.libquantum, and 470.lbm), with object counts ranging from just one object for 462.libquantum, to 40 objects in 433.milc. Thus, workloads with a lower number of objects have low cache-sensitivity.

Contrary to that, benchmarks such as 483.xalancbmk, 473.astar, 464.h264ref, and 403.gcc exhibit a clear trend, wherein the slowdown increases as miss rate rises. The object counts for these workloads (See Figure 4 for details) increase above the largest cache sizes evaluated. But, a high object count alone does not predict performance slowdown, as seen in Figure 4. For example, the slowdown for 473.astar remains near 65% for all cache sizes. As the miss rates remain constant (near 20%), the overall performance impact for this workload is modest. Nevertheless,

for 483. `xalancbmk`, a much smaller miss rate (between 1% and 5%) translates to a much higher slowdown (from 52% to 240%). This is likely caused by 483. `xalancbmk`'s sensitivity to the misses – these object loads are critical to application progress. The 473. `astar` application, in comparison, is much less susceptible to its higher miss rate, as the out-of-order processor is able to hide the additional latency of these misses. This is a clear demonstration into why a detailed simulation methodology is needed, as using GREENFAT cache miss rates alone does not translate to an accurate performance impact of this implementation.

For a 64KiB cache, the overhead of hardware GREENFAT implementation is 8.46% compared to 11.19× for the software BLUEFAT implementation. We caution that these results are for slightly different benchmarks, so a direct one-to-one comparison is not possible. Nevertheless, it is clear that GREENFAT represents a significant performance improvement, since major overheads (e.g., DBI) are avoided altogether. Finally, we remark that our GREENFAT implementation represents a relatively conservative hardware extension, in the form of an additional cache and pointer decode step. Although this is not zero-overhead, our approach is compatible with existing conventional microarchitectures without a radical redesign.

## 5.4 Limitations

FRP is designed to prevent attacks under stronger threat models including brute force attacks (see Section 2.2). FRP does not prevent other attacks from outside of this model. That said, most exploits depend on a memory error as part of the attack chain, and traditional memory errors still represent a significant attack surface. Another limitation is that, like other binary-only tools [7, 17, 40], BLUEFAT and GREENFAT can only protect heap objects. This is because stack/global object boundaries and lifetimes are generally ambiguous at the binary level. This is a general limitation, and is not specific to FRP. Similarly, objects allocated using *Custom Memory Allocators* (CMAs) will not be necessarily protected, or only partly protected, depending on the internal implementation of the CMA. Again, this is a general limitation not specific to FRP. These limitations could be lifted if our approach were to be ported to the source-level, or if the binary representation were to be augmented with the necessary information as a future extension.

FRP is compatible with most off-the-shelf binaries. Some binaries, such as Google Chrome [19], appear to be incompatible with DBI-based tools (including non-FRP tools) in general, so cannot be tested. Other binaries use custom tagged pointer representations that compete with our randomized pointer encoding. For example, the SpiderMonkey Javascript engine in Firefox [36] uses *NaN boxing* that packs data in the upper 16 bits of `x86_64` pointers. Such examples of incompatibility are rare, and the incompatibility likely affects *all* encoded-pointer defenses, and is not specific to FRP. Also, it may be possible to adapt the programs to remove such incompatibilities, by modifying the program at the source level as a last resort.

Any implementation of FRP must be careful not to introduce new attack vectors or side channels. This can be challenging in both software and hardware implementations. This is also a general problem that is not specific to FRP. That said, the high entropy of FRP may provide a buffer against some implementation-specific attacks that only leak partial information.

## 6 RELATED WORK

We briefly summarize the related work in this section.

### 6.1 Memory Error Defenses

For a relatively recent survey on memory error defenses, see [54].

*Memory poisoning.* As discussed, memory poisoning is a very common approach implemented by many tools [3, 6, 7, 17, 22, 23, 26, 40, 49, 51, 52]. The vulnerability to bypass attacks is well-known, and thus the technique is mainly used for bug detection rather than security hardening.

*Guard pages.* Another pre-existing idea is to insert inaccessible guard pages between memory objects, as used by *efence* [43], Archipelago [33], and GWP-ASAN. Accessing a guard page will trigger a memory fault (SIGSEGV). In addition to consuming large amounts of memory (both virtual and physical), the approach can be vulnerable to bypass in the absence of randomization.

*Use-after-free vulnerabilities.* There exist specialized use-after-free defenses [11, 38, 53, 56]. Oscar [11] similarly avoids reusing pointers to reallocated memory. Unlike FRP, Oscar relies on virtual memory tricks, specifically, mapping the same physical page into the virtual address space more than once. This has overheads, weaker security, and supports fewer reallocations until exhaustion.

*Pointer tagging.* Our approach shares some superficial similarities with *pointer tagging*. Here, unused pointer bits are repurposed for meta information, e.g., an object *id*. For example, HWAsan [50] for ARM tags each pointer with a randomized value that is checked against access. However, pointer tagging essentially encodes the *id* twice: once explicitly as the tag, and once more implicitly as the address. Our FRP encoding supports *orders of magnitude* more *ids* (40 versus 16 bits), enabling much stronger security.

*Fat pointers.* Another old idea is to replace pointers with a fat encoding that explicitly stores object metadata, as used by Safe-C [2], CCured [39], Cyclone [27], amongst others. Like FRP, this approach tracks pointer provenance and avoids pointer confusion. However, fat pointers violate (Rq.Word) and are generally not compatible with existing code without modification. Softbound [37] improves over fat pointers by separating metadata, but is not binary compatible and needs recompilation.

*Low-fat pointers.* Another idea is to compress fat pointers into a native address representation [14, 16]. Unlike FRP, this approach can still suffer from false positives in specific cases (e.g., pointer escapes).

*Associative maps.* Tools like CRED [46], MPX [24], and FuZZan [26] store metadata in a tree-based associative map. However, FRP is the first approach to use an associative map to implement full pointer randomization, with the corresponding security benefits.

*Probabilistic methods.* As discussed, DieHard [4] and DieHarder [42] are the most prominent examples of heap layout randomizers. However, such randomization is limited to the virtual address space region used by the heap, which is orders of magnitude smaller than the full FRP space. As such, heap layout randomizers have significantly weaker guarantees compared to FRP.

*Encrypted pointers.* PointGuard [10] *encrypts* pointers in memory to mitigate potential overwrites. Similarly, the `glibc` `setjmp` function mangles the `jmp_buf` structure, and Windows supports an `EncodePointer` API for encrypting pointers. However, these schemes are not general-purpose pointer encodings under (Rq.\*). For example, given an encoded  $p$ , then `DecodePointer(p+1)` yields a gibberish value.

Neither the PointGuard, `setjmp`, nor `EncodePointer` encodings are general-purpose memory error defenses. In contrast,  $C^3$  [29] is a probabilistic capability system based on *partial* pointer encryption but with lower entropy (see Table 1). We elaborate in Section 6.2.

*Pointer provenance.* The ambiguity between invalid and valid pointers is closely related to the notion of *pointer provenance* [34]. Some memory error defenses, such as fat pointers [39] and SoftBound [37], effectively implement a form of pointer provenance tracking by explicitly annotating pointers with object metadata. However, this tends to break binary compatibility, since this metadata must be explicitly tracked somehow.

We argue that our fully randomized encoding can be thought of as the first *implicit* provenance-preserving pointer representation. This is a probabilistic and emergent property: two FRP pointer values will have very different bit patterns, and thus it is extremely difficult for the two values to become inadvertently or maliciously confused. Furthermore, since FRP values are still ordinary machine words, there is no need for explicit object or provenance tracking that would otherwise break binary compatibility.

## 6.2 Randomization versus Authentication versus Encryption versus Capabilities

Our design uses *randomization* as opposed to other entropy-based methods, such as *authentication* or *encryption*. In this section, we briefly compare the different approaches.

Authentication-based methods use an address-based encoding augmented with a code or tag dependent on the address. Thus, if the address is (maliciously) changed, the code will no longer match allowing for the detection of the error. The code usually occupies otherwise-used address bits (e.g., upper 24 bits under some PAC configurations), so is a relatively simple architectural change. However, the problem is that the *address itself must still occupy the other 40 bits unchanged*—fundamentally limiting an entropy-based defense.

Encryption-based methods are similar to randomization. The main difference is that randomization uses a *dynamic* bijection (represented by  $\mathcal{M}$  in FRP) between pointer values and objects, whereas encryption-based methods use a *static* bijection determined by a hidden key. We carefully considered an encryption-based design, as it eliminates the need to maintain the dynamic map  $\mathcal{M}$  altogether, which is a significant simplification. However, encryption has a significant weakness regarding *use-after-free* (UaF). Specifically, if an object is `free`d and reallocated with the same underlying address, then the cipher text (the encrypted pointer) will also be the same. The  $C^3$  system adds a mitigation for this problem by including an 8 bit *version* bitfield into the plaintext, meaning that each address has  $2^8=256$  encrypted pointer variants. However, under our stronger threat model, this equates to only **8 bits** of effective security, significantly weaker than the 24 bits for other classes of memory error. By using a dynamic mapping, FRP maintains the full (52 bit) security regardless of the error class.

Finally, our approach has similar properties to capability-based systems, including CHERI [55] and Capstone [57]. These systems replace the notion of addresses and pointers with *capabilities*, which can be thought of as a permission to access (or do some other operation on) some underlying object. Capability systems carefully track *provenance* as well as protect against capability *forging* or *tampering*. Like  $C^3$ , FRP offers similar protections but by using a probabilistic-based methodology. Traditional capability-based systems, such as CHERI [55], require a fundamental hardware redesign as well as porting the entire software stack, and hence, do not have compatibility as a goal. In contrast, FRP is binary-compatible with most existing software, and can be implemented with a relatively small hardware change in the form of an additional cache. Furthermore, FRP achieves much stronger protection compared with other probabilistic-based capability systems such as  $C^3$ .

## 7 CONCLUSION

Over the decades there has been significant research into memory error defenses. Despite this, there is no universally adopted (“always-on”) solution, as of the time of writing. The underlying problem is that each solution has inherent limitations and trade-offs, especially in relation to performance, compatibility, and security (bypass resistance). Our view is that recent hardware and entropy-based defenses (such as pointer randomization, authentication, or encryption) are the most promising in terms of performance and compatibility. However, the effective security of these existing defenses is still relatively low (e.g., 24 bits), already leading to some practical bypass attacks, especially against stronger threat models that can retry (“brute force”) attacks. Experience has shown that

such attacks are not unrealistic, meaning that even stronger defenses are needed in order to provide meaningful security.

In this paper, our aim was to defend against even stronger attack models, including the powerful ability to repeat attack attempts. Our approach was to design a new pointer encoding that is strongly *decoupled* from the underlying memory address, meaning that the pointer values can be fully randomized without affecting the underlying layout of objects within memory—a.k.a. *Fully Randomized Pointers* (FRP). Essentially, the FRP approach is to represent pointers as (cryptographically secure) random numbers. We show that the FRP encoding is very difficult to bypass, since it removes the ability to predict the value of target pointers necessary in order to execute a successful attack. Furthermore, the very high degree of randomization counters any attempt to “brute force” the necessary value. We evaluate our design against a strong-yet-realistic attacker model, and show that our approach has a very high attack resistance and cannot be easily bypassed.

We have implemented FRP in the form of a software prototype (BLUEFAT) as well as a hardware simulation (GREENFAT). BLUEFAT is built on top of a *Dynamic Binary Instrumentation* (DBI) framework, allowing for encoded pointers to be decoded and checked on-the-fly. We show that BLUEFAT provides strong security and is highly compatible, despite using a radically different pointer encoding, and has comparable performance to existing DBI-based memory error detection tools. For general purpose use of FRP, we also evaluate a proof-of-concept hardware simulation (GREENFAT) based on a modified load-store pathway. This allows for a hardware implementation of the cache, resulting in a practical performance overhead of 8.46%

## REFERENCES

- [1] Andrea A. Bittau, B. Adam, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking Blind. In *Security and Privacy*. IEEE, 2014.
- [2] T. Austin, S. Breach, and G. Sohi. Efficient Detection of All Pointer and Array Access Errors. 1994.
- [3] J. Ba, G. Duck, and A. Roychoudhury. Efficient Greybox Fuzzing to Detect Memory Errors. In *Automated Software Engineering*. ACM, 2022.
- [4] E. Berger, , and B. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Programming Language Design and Implementation*. ACM, 2006.
- [5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.
- [6] D. Bruening and Q. Zhao. Practical Memory Checking with Dr. Memory. In *Code Generation and Optimization*. IEEE, 2011.
- [7] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *International Symposium on Code Generation and Optimization*,, 2011.
- [8] J. Bucek, K. Lange, and J. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *International Conference on Performance Engineering*. ACM, 2018.
- [9] Chromium. Memory safety in chromium security, 2020.
- [10] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Security Symposium*, Washington, D.C., 2003. USENIX.
- [11] T. Dang, P. Maniatis, and D. Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Security Symposium*. USENIX, 2017.
- [12] S. Dinesh, N. Burow, D. Xu, , and M. Payer. RetroWrite : Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *Security and Privacy*. IEEE, 2020.
- [13] G. Duck, G. Xiang, and A. Roychoudhury. Binary Rewriting without Control Flow Recovery. In *Programming Language Design and Implementation*. ACM, 2020.
- [14] G. Duck and R. Yap. Heap Bounds Protection with Low Fat Pointers. In *Compiler Construction*. ACM, 2016.
- [15] G. Duck and R. Yap. EffectiveSan: Type and Memory Error Detection using Dynamically Typed C/C++. In *Programming Language Design and Implementation*. ACM, 2018.
- [16] G. Duck, R. Yap, and L. Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *Network and Distributed System Security Symposium*. Internet Society, 2017.

- [17] G. Duck, Y. Zhang, and R. Yap. Hardening Binaries against More Memory Errors. In *European Conference on Computer Systems*. ACM, 2022.
- [18] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. Halderman. The Matter of Heartbleed. In *Internet Measurement Conference*. ACM, 2014.
- [19] Google. Google Chrome Web Browser, 2021.
- [20] Ali Hajiabadi, Andreas Diavastos, and Trevor E. Carlson. Noreba: A compiler-informed non-speculative out-of-order commit processor. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 182–193, New York, NY, USA, 2021. Association for Computing Machinery.
- [21] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction-Level Parallelism*, 7:1–28, 09 2005.
- [22] N. Hasabnis, A. Misra, and R. Sekar. Light-weight Bounds Checking. In *Code Generation and Optimization*. ACM, 2012.
- [23] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Winter Conference*, 1992.
- [24] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer’s Manual, 2023.
- [25] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. International Organization for Standardization, 2018.
- [26] Y. Jeon, W. Han, N. Burow, and M. Payer. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. In *Annual Technical Conference*. USENIX, 2020.
- [27] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Annual Technical Conference*. USENIX, 2002.
- [28] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *Security and Privacy*. IEEE, 2019.
- [29] M. LeMay, J. Rakshit, S. Deutsch, D. Durham, S. Ghosh, A. Nori, J. Gaur, A. Weiler, S. Sultana, K. Grewal, and S. Subramoney. Cryptographic Capability Computing. In *Microarchitecture*. ACM, 2021.
- [30] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *Computer and Communications Security*. ACM, 2022.
- [31] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kanoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikolieris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+, 2020.
- [32] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation*. ACM, 2005.
- [33] V. Lvin, G. Novark, E. Berger, and B. Archipelago: trading address space for reliability and security. In *Architectural Support for Programming Languages and Operating Systems*. ACM, 2008.
- [34] K. Memarian, V. Gomes, B. Davis, S. Kell, A. Richardson, R. Watson, and P. Sewell. Exploring C Semantics and Pointer Provenance. *Proceedings of the ACM on Programming Languages*, 2019.
- [35] Microsoft. Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape, 2019.
- [36] Mozilla. Firefox browser, 2021.
- [37] S. Nagarakatte, Z. Santosh, M. Jianzhou, M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Programming Language Design and Implementation*. ACM, 2009.
- [38] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *International Symposium on Memory*. ACM, 2010.
- [39] G. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. 2002.
- [40] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Programming Language Design and Implementation*. ACM, 2007.
- [41] NIST. Juliet Test Suite for C/C++ v1.3, 2022.
- [42] G. Novark and E. Berger. DieHarder: Securing the Heap. In *Computer and Communications Security*. ACM, 2010.
- [43] B. Perens. Electric Fence Malloc Debugger, 2022.

- [44] J. Ravichandran, W. Na, J. Lang, and M. Yan. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In *Computer Architecture*, page 685–698. ACM, 2022.
- [45] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebe. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Operating Systems Design & Implementation*. USENIX, 2004.
- [46] O. Ruwase and M. Lam. A Practical Dynamic Buffer Overflow Detector. In *Network and Distributed System Security Symposium*. Internet Society, 2004.
- [47] Alen Sabu, Harish Patil, Wim Heirman, and Trevor E Carlson. Looppoint: Checkpoint-driven sampled simulation for multi-threaded applications. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 604–618. IEEE, 2022.
- [48] G. Saileshwar, R. Boivie, T. Chen, B. Segal, and A. Buyuktosunoglu. HeapCheck: Low-Cost Hardware Support for Memory Safety. *Transactions on Architecture and Code Optimization*, 19(1), 2022.
- [49] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Annual Technical Conference*. USENIX, 2012.
- [50] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyklevich, and D. Vyukov. Memory Tagging and how it Improves C/C++ Memory Safety. In *Hot Topics in Security*. USENIX, 2018.
- [51] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Annual Technical Conference*. USENIX, 2005.
- [52] K. Sinha and S. Sethumadhavan. Practical Memory Safety with REST. In *Computer Architecture*. IEEE, 2018.
- [53] E. Kouwe and V. Nigade and C. Giuffrida. DangSan: Scalable Use-after-free Detection. In *European Conference on Computer Systems*. ACM, 2017.
- [54] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for Security. In *Security and Privacy*. IEEE, 2019.
- [55] R. Watson, J. Woodruff, P. Neumann, S. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Security and Privacy*. IEEE, 2015.
- [56] Y. Younan. FreeSentry: Protecting Against Use-after-free Vulnerabilities due to Dangling Pointers. In *Network and Distributed System Security*. Internet Society, 2015.
- [57] J. Yu, C. Watt, A. Badole, T. Carlson, and P. Saxena. Capstone: A Capability-based Foundation for Trustless Secure Memory Access. In *Security Symposium*. USENIX, 2023.