SoK: On the Fragility of Memory Error Exploit Mitigations

Adriaan Jacobs*
DistriNet, KU Leuven

Mahmoud Ammar*
Huawei Research, Germany

Stijn Volckaert
DistriNet, KU Leuven

Abstract

The perennial war in memory has long been shaped by a continuous arms race: defenses are deployed, bypasses emerge, and stronger mitigations follow, only for the cycle to repeat. This pattern persists in part due to the *fragility* of many defenses, which often *fail* when assumptions change. Such fragility reflects how security guarantees are assessed, often through ad hoc reasoning tied to specific threat models. In a fast-evolving landscape of polyglot applications and heterogeneous systems, manually re-evaluating these guarantees for every new context is both labor-intensive and error-prone.

This SoK advocates for a more systematic, adversary-aware approach. We introduce a graph-based framework for evaluating the fragility of memory safety defenses by modeling the progression of memory corruption exploits, and where, how, and under what assumptions defenses intervene and may fail. We demonstrate the utility of this model by revisiting flaws in prominent defenses and showing how their shortcomings could have been anticipated. Finally, we release open-source tooling that implements our model and supports systematized and semi-automated fragility testing.

1 Introduction

Memory corruption errors in unsafe programming languages like C and C++ have long been at the forefront of software exploitation [221], dating back to the Morris Worm [208], which exploited a buffer overflow vulnerability over three decades ago. Since then, a multitude of exploit mitigations have been proposed, ranging from vulnerability-centric defenses, such as bounds checking [5, 67, 88, 159, 233, 236, 243], type miscast detection [17, 96, 110, 131], and temporal safety mitigations [3, 78, 79, 160, 231], to more exploit-centric approaches, including basic memory protection features (e.g., Data Execution Prevention (DEP) [154] and stack canaries [138]), code randomization strategies [58, 98, 117, 179, 184], Control Flow Integrity (CFI) [1, 80, 102, 213], Data Flow Integrity (DFI) [4, 38, 205, 228], among others [37, 45, 132, 144, 226, 230].

However, despite sustained efforts to advance memory safety defenses, the battle against memory corruption remains intense and far from resolved [140], with evolving exploitation techniques continually challenging existing measures [35, 39, 51, 100, 101, 107, 109, 113, 143, 152, 202]. This enduring struggle persists in part because, even as exploits and defenses have grown more complex, our core approaches to analyze and characterize them have changed little since the early days of stack overflows and shellcode injection [173].

Manual security analysis based solely on human expertise has historically been unreliable. Memory error mitigations, in particular, have shown to be remarkably *fragile*, not merely due to implementation bugs, but because they often fail to anticipate the breadth of exploitation techniques enabled by memory errors.

> We define **fragility** as a defense's susceptibility to failure when assumptions about its deployment scenario, threat model, or interaction with other components or defenses are violated or only partially hold.

This fragility is evident in macro-level exploitation trends, from shellcode injection to code reuse attacks [29, 187]—in all their subversive forms [35, 73, 194]—and data-only attacks [100, 101, 107, 113]. It is also visible in individual defenses [51, 76, 77, 109, 135] and even entire classes of mitigations [35, 189, 202]. For instance, despite extensive research activity, it took over a decade to show that code randomization alone could not stop code reuse attacks [189].

The challenge is further exacerbated by the growing heterogeneity of the computing ecosystem, in which computing units provide different baseline security features on which defenses implicitly depend. For instance, modern accelerators, such as Nvidia GPUs, are likely susceptible to memory errors similar to those observed in CPUs [64, 90, 155, 186], yet still lack even basic memory protection features [90]. Such capabilities have been assumed as standard on CPUs for over two decades [240], raising the question of whether mitigations

*Equal contribution joint first authors.

designed for CPUs transfer to other platforms like GPUs and NPUs where key assumptions may no longer hold.

Even within CPU environments, defenses are frequently deployed under conditions that differ from those assumed during design and evaluation. Multi-language applications that combine memory-safe components with hardened memory-unsafe ones illustrate this issue [152, 176]. For instance, Mergendahl et al. [152] showed that rewriting parts of a CFI-hardened C or C++ codebase in Rust created a gap through which control-flow hijacking could succeed. Although Rust and CFI independently defend against such attacks, overwriting a Rust code pointer from unsafe code can restore the attack vector and bypass both protections.

Collectively, these cases reveal a pronounced sensitivity to minor adaptations and a design-level *fragility* under shifting assumptions, which is especially undesirable for mitigations intended to protect the most security-sensitive software projects in the computing stack [50]. In essence, this issue arises because the security guarantees of many defenses remain insufficiently characterized under a unified and representative threat model that can expose *design-level* gaps when such defenses are evaluated under varying conditions. As a result, it often remains unclear, on the one hand, how these defenses interact with others and whether their interplay introduces security gaps; and, on the other hand, whether their security guarantees hold under diverse deployment scenarios.

A scalable and effective approach to characterizing the security guarantees of memory safety defenses, particularly to identify their *design-level fragility* under specific conditions, should begin with a systematic and attacker-aware perspective. To this end, **we propose to model memory error exploitation as a graph reachability problem**. We present a graph-based framework grounded in an attack-centric model that captures the full spectrum of adversarial techniques, from initial exploit primitives to post-compromise objectives. In our framework, vertices represent states or adversarial capabilities, while edges denote actions taken by adversaries to transition between states or acquire new capabilities. These transitions form pathways that culminate in concrete means, such as code injection or control-flow hijacking, to achieve malicious objectives, such as arbitrary execution control.

We then map defenses to specific exploitable edges, identifying where each defense takes effect. If an adversary's goal remains reachable despite a defense, this indicates a design gap or a sign of fragility under the evaluated scenario. Using this model, we systematically map a wide range of memory corruption defenses onto the graph, yielding a machine-analyzable and visual representation of effectiveness across deployment settings and in combination with other mitigations. We demonstrate the utility of our model through distinct use cases that uncover where defenses interact, where they fall short, and how their effectiveness varies under realistic conditions. Finally, we release open-source tools that implement our model and support semi-automated fragility testing.

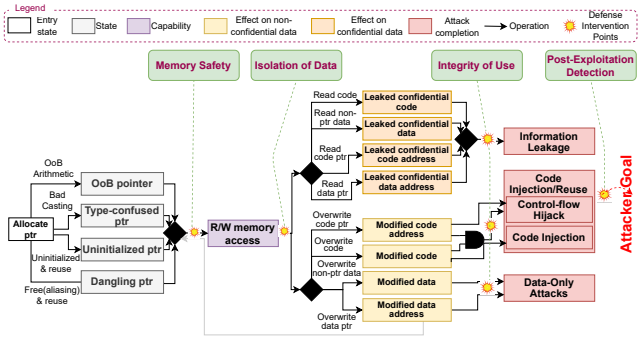

Figure 1: Typical flow of memory corruption attacks (as proposed by Szekeres et al. [211]) and intervention points of defense approaches (as characterized in this work).

2 Setting The Stage

This section revisits how prior work has modeled memory corruption exploits and defenses, using the seminal SoK by Szekeres et al. [211] as a running example. We then show how a *layered* view of attack flow fragments defenses into imperfect categories and provides limited insight into the fragility of defenses, as defined in Section 1.

2.1 Flow of Memory Corruption Exploits

Figure 1 summarizes the typical progression of memory corruption attacks. Exploits typically start from an **invalid pointer**, resulting from out-of-bounds (OoB) arithmetic, bad casting, uninitialized memory, or dangling references to freed objects. Such corrupted pointers give the adversary unauthorized read (R) and/or write (W) access to memory, which enables leaking sensitive information (e.g., cryptographic keys in Heartbleed [72]), corrupting sensitive data to escalate privileges [39], or hijacking control or data flow to subvert program behavior [29, 101, 187]. End-to-end exploits are often grouped into four categories:

- **Information Leakage:** Exploits that rely on unauthorized reads to disclose secrets such as cryptographic keys [72], memory layouts [179], or pointer values [204], which could further facilitate other attacks.
- **Code Injection:** Attacks that exploit unauthorized write accesses to inject attacker-controlled code or payloads into executable memory regions and then execute them [24].
- **Code Reuse attacks:** Techniques such as Return-Oriented Programming (ROP) [187] and Jump-Oriented Programming (JOP) [29] that redirect a program's control flow using existing code sequences without injecting any new code.
- **Data-only attacks:** Exploits that manipulate non-control data to alter program behavior without violating its control flow integrity [45], including Direct Data Manipulation (DDM) [39] and Data-Oriented Programming (DOP) [101].

2.2 Defense Strategies and The Missing Link

We now derive four recurring defense strategies from the literature to better characterize what guarantees defenses aim to provide and under which conditions these guarantees may become fragile. We then show, via concrete examples, that a strategy-level view alone is still too coarse when paired with an abstract model of memory corruption attacks, motivating the finer-grained model introduced in Section 3.

2.2.1 Exploit Mitigation Strategies

As illustrated in Figure 1, we find that exploit mitigations tend to follow four high-level strategies, each intended to probabilistically or deterministically disrupt the adversary's progress at a specific stage of the exploitation lifecycle.

Memory Safety Enforcement Memory Safety Enforcement aims to mitigate memory errors by enforcing that memory is accessed according to intended semantics. This requires preventing *spatial errors* (accesses beyond object bounds), *temporal* errors (dereferencing deallocated or otherwise invalid objects), and *type* errors (misinterpreting an object's type). Enforcing full or partial memory safety stops the attack early by preventing the use of invalid pointers.

A wide range of defenses aim to enforce memory safety at varying levels of completeness and overhead. Languages such as Rust [151] provide spatial, temporal, and type safety through compile-time checks and ownership rules. Retrofitting similar guarantees into legacy C/C++ code requires extensive instrumentation at the source [111, 163], Intermediate Representation (IR) [5, 159], or binary level [68, 95]. Spatial safety is typically enforced through pointer- [88, 111, 159, 233] or object-based [5, 115, 162, 242] bounds checking approaches, as well as *hybrid* ones that combine aspects of both, often for better compatibility or performance [108, 161, 197]. Temporal safety is tackled by altering allocation policies [104, 142], quarantining freed memory [164, 167, 198], or associating metadata with pointers to detect use-after-free and uninitialized accesses [160]. Type safety is often enforced through run-time checks on casting operations [17, 96, 110, 131].

Isolation of Data (IoD) This strategy assumes an adversary with arbitrary R/W memory access and aims to prevent the exploitation of such access by protecting sensitive code or data from corruption or leakage. Approaches following this strategy fall into two paradigms: deterministic *isolation* and probabilistic *hiding*. Isolation mechanisms establish strong boundaries to shield critical memory regions, limiting the attacker's ability to access or manipulate them. These include pure software techniques such as Software Fault Isolation (SFI) [226], as well as hardware-assisted schemes that leverage features like Intel Memory Protection Keys (MPK) [44, 112, 218] or ARM Memory Tagging Extensions (MTE) [13, 139, 196]. In contrast, hiding mechanisms obfuscate the representation or location of sensitive code and data. ISR [117] protects against code injection by altering instruction encodings, whereas ASLR [179] disrupts code reuse by randomizing memory layouts [129]. DSR [184] further mitigates data-oriented and memory disclosure attacks by encoding program data using secret keys. Code Pointer Integrity (CPI) and its relaxed variant, Code Pointer Separation (CPS) [127], use either isolation or hiding to protect code pointers from corruption, depending on available hardware support.

Integrity of Use (IoU) This strategy focuses on ensuring that, even if certain code or data elements (e.g., code or data pointers) are corrupted or leaked, the adversary cannot use them to trigger malicious behavior such as arbitrary code execution. IoU mechanisms enforce correct program behavior by constraining how data elements are used during execution, typically by comparing it to statically or dynamically collected *evidence*. Representative examples include CFI [1, 8, 49, 102, 119, 147, 149, 165, 172] and DFI [4, 38, 45, 205, 228], which typically compare data elements such as code or data pointers against statically-built sets of valid values. Multi-Variant Execution (MVX) [56, 191, 192, 222] instead compares execution behavior among diversified program variants, and uses divergences as dynamic evidence of an attack.

Post-Exploitation Detection This strategy represents the final line of defenses [45]. It aims to detect successful exploits that have evaded all prior protection mechanisms [9], by regularly attesting and verifying the integrity of binary code [7, 168] and runtime state [8, 169, 170].

2.2.2 From Fragmentation to Connection

The layered model in Figure 1, inspired by Szekeres et al. [211], is useful for high-level classification, but forces defenses into rigid categories that fail to capture their nuanced defensive properties. In practice, many defenses defy rigid categorization and span multiple strategies. WIT [4], for instance, compares the targets of data pointers in write operations with a statically built set of valid values, which simultaneously enforces IoU for data pointers and IoD for all program data, with a single check. Likewise, CFA+ [8] combines elements of IoU with post-exploitation detection to counter code reuse attacks. Also MVX systems, when combined with DSR schemes, may implicitly enforce data isolation [124,222] (IoD-style), providing stronger guarantees than suggested by the IoU policy.

Furthermore, the layered abstraction models exploitation as a mostly linear progression, while realistic exploits often repeatedly escalate attacker knowledge and exploitation primitives over multiple triggered vulnerabilities. Cyclic dependencies between attacker primitives are particularly relevant to model probabilistic defenses, which may partly depend on the secrecy of values in memory (e.g. code or data addresses), and may be defeated via exploratory information disclosure attacks [209]. To evaluate the effectiveness of "leakage-resilient"

probabilistic defenses [21, 189], that attempt to deterministically inhibit such leakage, these cyclic dependencies must be precisely modeled. Other attacks may also depend on cycles in the exploit graph, e.g., a Data-Oriented Programming (DOP) [101] attack where corrupted non-control data redirects direct control flow to introduce new first-level exploit primitives (Section 3.4).

In Section 3, we describe how our model overcomes these limitations, among others, to enable precise mapping of defenses to attack steps and systematic reasoning about assumption violations and defense fragility.

3 Anatomy of a Memory Error Exploit

A model of memory error exploits must be both *detailed* and *complete* to provide trustworthy insights into the *fragility* of mapped defenses. It needs to be *detailed* because subtle differences in the capabilities permitted by defenses can affect their interoperability. For instance, pointer-based bounds checkers [159] allow out of bounds (OoB) pointers but check them at dereference time, whereas object-based ones [69, 115] prevent pointers from leaving their referent. When these mechanisms interact, an object-based checker may incorrectly treat a pointer from a pointer-based system as valid. Similar mismatches arise with conservative garbage collectors [3, 30] or dangling pointer scanners [130, 219, 241], which may misidentify temporarily OoB pointers as references to different objects [79], enabling unsafe reuse. To capture such nuances, our model introduces edges and capabilities as needed to represent a broad and growing set of defenses.

At the same time, the model must be sufficiently *complete* to justify claims that particular attack vectors are mitigated, even though absolute completeness is unattainable [71]. Centralizing and structuring known attack vectors in a single model remains a practical step towards this goal. In the rest of this section, we introduce the model following a topic-centered approach, presenting the primitives required to construct a memory corruption attack and how they compose into Figure 2.

3.1 Assumptions, Terminology, and Concepts

We first survey a broad range of exploitation techniques that target memory corruption on CPUs, as reported in academic literature [24, 26–29, 35, 36, 39, 40, 42, 43, 45, 51, 60, 71–73, 75–77, 91–93, 98, 100, 107, 109, 113, 129, 141, 146, 148, 152, 173, 176, 177, 182, 185, 187–189, 193, 194, 201, 202, 204, 208, 209, 223, 234, 240] and at industrial and hacking conferences [114, 195, 200], as well as recent attacks on AI accelerators such as GPUs [64, 90, 155, 178, 186]. Our goal is to encode the adversarial capabilities exercised in these exploits into a unified and representative model. We also review a large body of industrial [10–12, 87, 144, 151, 153, 154, 180, 203, 213, 215] and academic defenses [1, 2, 4–6, 8, 14–19, 21–23, 25, 34, 37, 38, 41, 44, 46–48, 53, 56–59, 61–63, 65, 67–70, 74, 78–84, 86, 88, 94, 96, 102–106, 110–112, 116, 117, 122–127, 130, 131, 133, 135–139, 142, 145, 157–160, 162, 163, 166, 171, 174, 175, 177, 179, 183, 184, 190–192, 197–199, 205, 206, 212, 217–220, 222–231, 235, 236, 239, 241, 243–245], including the CVEs and attack patterns they explicitly target, as well as vulnerability benchmarks and test suites [54, 55]. The outcome of this analysis is the graph-based framework shown in Figure 2.

Format. We introduce a graph-based format (Figure 2) in which vertices denote adversarial *capabilities* and edges represent *actions* taken by adversaries to unlock new capabilities. Defenses typically protect specific edges to prevent further adversarial progress (see Section 5). This detailed format exposes the differences in attack surfaces between various defenses. For instance, a late-stage defense like CFI [32] is meaningful only after an attacker has already gained some capabilities, whereas an early-stage approach like SoftBound [159] attempts to prevent such capabilities from arising.

Threat Model. We consider an adversary with full access to the application source code, who can execute the program arbitrarily often to discover memory errors and derive exploit primitives. The model assumes complete attacker expressiveness by default, and requires all mitigations to be stated explicitly (including "common" ones like DEP, ASLR, stack canaries, or CFI), to serve as a neutral baseline. We focus on generic consequences of memory corruption and the capabilities they enable, rather than application-specific policies. In Section 3.4, we introduce a general way to model such policies.

Target Elements. We target four distinct elements of the programming environment: executable code, code pointers (pointers to executable code), data pointers (pointers to data), and non-pointer data (Plain Old Data). These cover all memory objects involved in typical attacks. Among code pointers, we distinguish forward-edge pointers (function pointers) from backward-edge pointers (return addresses), because defenses often treat them differently and return addresses obey stricter semantics that preclude some attack vectors [109]. These four elements form the foundation for characterizing the memory corruption attack surface.

Knowledge. Exploitation also depends on the adversary's knowledge of program layout and contents. Defenses frequently try to frustrate attacks by restricting this knowledge [58, 128, 223]. We model this through four knowledge capabilities, which randomization-based defenses may inhibit:

- **Code Layout Knowledge (CLK)** refers to the knowledge of the location of executable code in memory, needed to redirect control flow to valid instructions despite coarse- or fine-grained code randomization mechanisms [22, 26, 57, 58]. CLK is often obtained by leaking code pointers [57].
- **Code Content Knowledge (CCK)** captures whether the adversary knows the instruction-level content of a code region, which is needed to construct gadget-based reuse attacks even when the victim's binary is randomized at the

sub-function level [16,61,62,98,120,177,229]. Unlike CLK, CCK generally requires direct disclosure of code [28, 204], motivating defenses against code leakage [15, 212, 244].

- **Data Layout Knowledge (DLK)** models whether an adversary can determine the locations of objects in memory. It is targeted by data layout randomization schemes at various granularities [26, 52, 124, 227] and is usually gained by leaking data pointers [202, 209].
- **Data Content Knowledge (DCK)** refers to the understanding of the semantics of non-pointer data, such as how credentials or policy values are represented in memory [141]. DCK is rarely constrained by existing mitigations.

Each knowledge type governs specific exploitation steps. For example, overwriting a code pointer via buffer overflow may require DLK to locate the target pointer and CLK to write a useful corrupted value. In our model (Figure 2), this dependency appears as an **AND** relation between required knowledge and the exploited capability edge.

3.2 Gaining Arbitrary Memory Access

To gain arbitrary memory access, an adversary must first invalidate a vulnerable pointer, following one of the pointer invalidation techniques shown in the upper part of Figure 2 (1 - 13). By pointer invalidation we mean bugs that cause pointers to violate their intended aliasing rules and reference attacker chosen memory instead of their original referent.

Spatial errors occur when pointers leave the bounds of their referent [115], while temporal errors arise when pointers outlive the lifetime of their referent and become dangling [79]. Both classes enable the adversary to redirect pointers away from their intended object. **Spatial errors** typically grant the ability to add chosen offsets to a pointer 3 and, given DLK, to redirect it to specific victim objects. **Temporal errors** can be exploited through memory massaging, where freed regions are intentionally reused for attacker chosen objects [121]. In Figure 2, we model dangling pointers that may arise from the combination of allocator memory reuse 2 and the presence of aliasing pointers 6 , both of which can be inhibited by various defenses [79, 130, 160, 219].

Two additional bugs also produce invalid pointers. First, **type confusion** has emerged as a popular source of 0-day vulnerabilities [87]. It arises from invalid casts 4 between unrelated types in C [96, 131] or from unchecked downcasts in C++ [17, 110]. Most importantly, these miscasts can make attacker-controlled non-pointer data be interpreted as a valid pointer, enabling pointer forgery and arbitrary memory access similar to OoB or dangling pointers [27, 110]. Second, correctly allocated and typed pointers can reference **uninitialized memory** due to reuse of memory that was not fully overwritten [175]. By manipulating memory layout, an attacker can influence what data appears at such reused addresses, leaking secrets or controlling read values. Pointers to uninitialized memory can only grant invalid reads (1), yet such bugs still account for about 10% of recent 0-day vulnerabilities [94].

3.3 Abusing Arbitrary Memory Access

Given an invalidated pointer capability, adversaries typically dereference it to gain arbitrary memory read or write access. We model reads and writes as attacker-controlled data flows from a source to a destination. For reads, the source is attacker-chosen memory and the destination is a register or variable. For writes, the source is a variable and the destination is attacker-chosen memory. Based on this, we construct a matrix of data or memory types involved in these flows (Tables 1 and 2). We limit our framework to data flows that can occur benignly during execution yet are influenceable by adversaries. This captures the first exploitation step where a memory error manifests.

3.3.1 Abusing Read Access Capabilities

Read Leakage. The most common CVEs involving memory read vulnerabilities stem from invalid manipulations on pointers to non-pointer data objects, such as buffer over-reads during string operations, as in Heartbleed [72]. These are illustrated as **Leak** actions stemming from the **READ** capability in Figure 2 (4 edges: 1 - 4): confidential data is invalidly read into an application output buffer and later sent over a legitimate output channel [19].

Read Substitution. Independently of the victim object type, invalid reads can substitute objects with other objects of the same type: the program typically interprets the loaded value according to the expected type rather than the actual memory contents. This exploitation pattern has been understood for many years [211] but was overlooked by an entire line of research on store-only bounds checkers, which assumed checking writes alone would prevent memory corruption [109]. Such an oversight illustrates how manual, assumption-based reasoning about defenses can miss important attack paths. Figure 2 shows all possible **Substitute** actions originating from the **READ** capability (3 edges: 8 - 10).

Read Crafting. Illustrated as **Craft** actions stemming from **READ** in Figure 2 (3 edges: 5 - 7), this exploitation pattern arranges that a read from attacker-controlled non-pointer data, such as user-supplied payloads, is interpreted as a pointer or structured object. This produces *crafted* values that the application later treats as benign [109, 110].

Table 1 summarizes the three distinct forms of exploitation arising from invalid memory reads, as visualized also as corresponding actions stemming from the **READ** access capability in our model (edges from 1 to 10 in Figure 2).

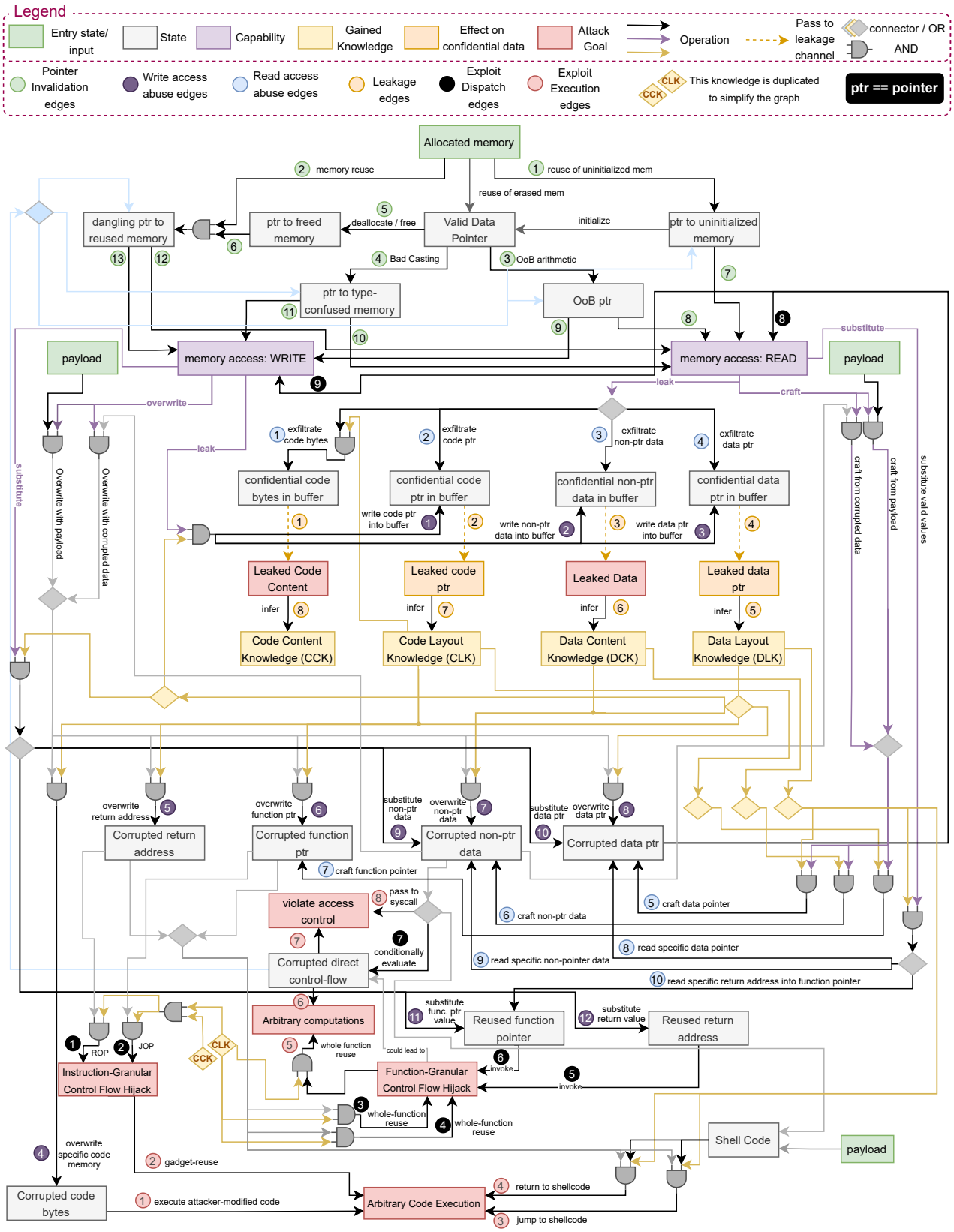

Figure 2: Visual representation of our model of memory corruption attacks: from pointer invalidation to post-compromise goals.

Table 1: Possible effects of invalid memory reads

Read value used as	Type of Read Memory: Code	Code Ptr	Data Ptr	Data
Code	–	–	–	–
Code Ptr	–	subst	–	craft
Data Ptr	–	–	subst	craft
Data	leak	leak	leak	*all*

Table 2: Possible effects of invalid memory writes

Type of Written Value	Overwritten Memory Used As: Code	Code Ptr	Data Ptr	Data
Code	–	–	–	–
Code Ptr	–	subst	–	leak
Data Ptr	–	–	subst	leak
Data	overwrite	overwrite	overwrite	*all*

3.3.2 Abusing Write Access Capabilities

Overwrites. The classical exploitation of arbitrary writes uses a *write-what-where* primitive [211]. It grants adversaries control to specify both the data to be written ("what") *and* the target memory location ("where"). All data types, including code bytes, code pointers, and data pointers, can be targeted, which makes this one of the most powerful and widely used primitives in 0-day exploits [87]. These **Overwrite** actions can be seen in Figure 2 as coming from the **WRITE** access capability (5 edges: 4-8).

Write Leakage. Arbitrary writes can also be used for data leakage, as an adversary may overwrite output buffers with sensitive values that the application will later expose. This requires the vulnerable write to source confidential data, which can be rare for application secrets such as cryptographic keys [112, 141], but can be substantially more common for other types of confidential data, such as randomized pointer values [19,97,109]. Figure 2 illustrates these **WRITE**-based **Leak** actions (3 edges: 1-3).

Write Substitution. Write substitution attacks arise when the vulnerable write copies a value of the same type as the victim object. For non-pointer data targets, the distinction between leakage, substitution, and overwrite largely reflects attacker intent: leakage writes confidential data to an output buffer; overwrite injects payload data into a security critical variable; and substitution reuses existing application data. For pointer targets, write substitution requires a pointer-typed source, which is less common than writes sourced from Plain Old Data such as strings. **WRITE**-based **Substitute** operations are shown in Figure 2 (4 edges: 9-12).

Table 2 summarizes the three different exploitation patterns of invalid writes: *overwrite*, *substitute*, and *leak*, and how they relate to the types of written values and overwritten memory.

3.3.3 Gained Adversarial Primitives

Regardless of whether exploitation starts from read or write capabilities, invalid memory accesses ultimately produce either corrupted or invalidly disclosed data. In practice, memory errors on non-pointer data, such as strings, are most frequent, so **WRITE** primitives tend to favor **Overwrite** actions, while **READ** primitives primarily favor **Leak**.

We conservatively treat all invalidly disclosed data as *confidential*, independent of its real world impact. Confidentiality in our model thus refers to provenance (valid benign flow versus invalid disclosure), not to whether the value is actually sensitive, since that depends on application-specific semantics. Under this abstraction, there are four leakage-related attacker primitives: each type of confidential data residing in output buffers (code, code pointers, data pointers, data). Additionally, there are four corresponding corruption-related types of attacker-corrupted variables in memory.

In the model, we explicitly distinguish between function pointers and return addresses as they adhere to different semantics. For instance, unlike function pointers, return addresses cannot be corrupted through read substitution under correct compilation. We also qualify the expressiveness of the obtained corrupted data type depending on source primitive. **Crafting** and **overwrite** can introduce values that never existed before, whereas **substitution** is limited to *reusing* existing values. This difference is most significant for code pointers: pointers to fine-grained gadgets are rarely present in memory [93, 126], so **substitution** attacks generally yield pointers to function entries or return sites. As reflected in Table 3, we model **substitution** as producing *reused* values that support only coarse-grained control-flow influence, such as Whole-Function Reuse [189, 194], rather than fine-grained gadget chaining as in ROP [187].

Table 3 enumerates which primitives become available after an initial invalid access and through which combination of **Leak**, **Substitute**, **Craft**, or **Overwrite** they arise. These edges (1-12,1-10) form the bulk of the complexity of the model in Figure 2.

3.4 Achieving Attacker Goals

With intermediate capabilities in hand, the adversary can work toward achieving higher-level (end) goals.

Leakage Goals. Any of the four data types can be leaked to gain specific **Knowledge** (Section 3.1). Leaking non-pointer data may yield sensitive information, such as credentials or secrets, which we model as **Leaked Data** in Figure 2; the Heartbleed attack [72] is a canonical example. Leaking code may yield Intellectual Property (IP) theft in the case of closed-source binaries [28] (**Leaked Code Content**) or defeat fine-grained code randomization [204] by revealing

Table 3: Primitives after invalid memory access

		READ			WRITE		
		Leak	Subst	Craft	Write	Subst	Leak
Corrupted	Code				✓		
	Func. Ptr.		reuse	✓	✓	reuse	
	Ret. Addr.				✓	reuse	
	Data Ptr.		✓	✓	✓	✓	
	Data	✓	✓	✓	✓	✓	✓
Leaked	Code	✓					
	Code Ptr.	✓					✓
	Data Ptr.	✓					✓
	Data	✓	✓	✓	✓	✓	✓

gadget locations. In all cases, confidential values must eventually be written to a leakage channel (1 - 4), which can still be blocked at this late stage by dynamic information flow tracking [118] or MVX systems that compare output buffers of variants with diversified secrets [20, 145, 223].

Corruption Goals. One risk stems from **corrupted code** . If executable memory can be modified, the program may execute attacker-controlled code 1 without any control-flow hijack. While such attacks are largely prevented on desktop and server systems by W⊕X policies [240], they remain possible on platforms without code write protection, such as many embedded and IoT systems and modern accelerators [90], as demonstrated by the Mind Control attack on GPUs [178].

Furthermore, **corrupted data pointers** enable attackers to trigger further memory errors 8-9, which is crucial in multi-stage exploits [100, 141] and for upgrading restricted bugs into more expressive primitives [40, 148, 234]. Moreover, **corrupted code pointers** remain some of the most attractive targets [1, 126] because they control indirect control flow transfers (iCFTs) and may lead to **Arbitrary Code Execution** (ACE). Early exploits simply overwrote code pointers to jump directly into data buffers containing injected code [173] 3-4. After the deployment of DEP [154], attackers shifted to code-reuse attacks such as ROP [187] or JOP [29] 2, which reuse existing victim code to implement the exploit [201] rather than injecting new code, often by chaining small snippets of victim code together, called gadgets.

Defenses that randomize code content to restrict CCK [16, 61, 62, 98, 120, 177, 229] and that prohibit direct code disclosure 1 [15, 244], or that enforce CFI and restrict iCFT targets to function entries [12, 57, 58, 153, 215], have driven attackers toward more coarse-grained Whole-Function Reuse (WFR) attacks [189, 194] (edges: 3-4).

Empirical analysis suggests that many exploits remain implementable using only WFR 5, especially in C++ applications with many small member functions [194]. Still, WFR cannot reach rare or unaligned gadgets [232] that some attacks depend on [122, 238]. In our model, data execution 3-4, direct corruption of code bytes 1, and fine-grained iCFT hijacks 2 all enable **Arbitrary Code Execution**. WFR, by contrast, cannot achieve ACE but is expressive enough to realize the higher level goal of **Arbitrary Computations** (AC) 5. This models the capability to force the victim program to implement arbitrary functionality, but without necessarily controlling the specific code sequence to do so.

This separation between ACE and AC also helps clarify nuances of data-only attacks. Initially defined as attacks that avoid code pointer corruption [39] and instead corrupt application-specific privilege-holding data [42, 100], data-only attacks were later shown by Hu et al. [101] to support Turing-complete interpreter-like loops in real programs, termed Data-Oriented Programming (DOP). The expressiveness of DOP depends on the available operations, but in practice, the presence of security-critical system calls with attacker-controlled arguments [100, 113, 239] makes DOP a viable route to **Arbitrary Computations** 6 in the absence of ACE [2, 82, 83, 210].

Thus, we introduce **corrupted direct Control Flow Transfer** (dCFT) as a separate attacker primitive: it arises from conditionally evaluating corrupted non-pointer data 7, which can violate application-specific access control policies and escalate privileges 7, or initiate a DOP-like Turing-complete interpreter ("gadget dispatcher") [101, 107] 6. Rather than modeling DOP as an explicit capability, we integrate its gadget dispatcher loop into the graph structure by adding edges from dCFT back to first-level pointer invalidation. The intuition is that control over decision-making data and direct control-flow can trigger logical inconsistencies, such as miscasts or bounds check bypasses, even if the original program is free of such bugs.

Takeaway-1: First-level memory vulnerabilities can be viewed as attacker-chosen swaps of data elements within a program. Vulnerabilities on different types of data, e.g., pointers/non-pointers, impose different constraints on the type and expressiveness of the leaked or corrupted data elements. Crucially, however, both **READ** and **WRITE** vulnerabilities can ultimately achieve similar outcomes: *leakage* and *corruption* of most data elements (Table 3). Underestimating the subversive power of **READ** vulnerabilities, particularly **Substitute** and **Craft**, introduces fragility that has been previously been used to bypass write-only exploit mitigations.

4 Mapping of Defenses

Our graph models over 50 defensible edges and four knowledge nodes, from which defenses can choose any combination to stop memory error exploits. We map 40 representative

Table 4: An overview of a broad set of unique and representative defenses, and how they map to our model.

Defense	Strategy	Category	Inhibited Edges or Knowledge	Assumed Threat Model	Attacker Goals			Execution Control		
					Info. Disclosure.	Access Control	Arbitrary Computation	Func.-Gran. CF Hijack	Instr.-Gran. CF Hijack	ACE
Safe Rust [150]	MS	Complete	1 3-5							
CCured [163]	MS	Complete	1 4-5 8-9							
SoftBound [159]	MS	Spatial	8-9 5 7 5 3 2	1 4-5						
SoftBound-w.o. [159]	MS	Spatial	9 5 7 5 3 2	1 4-5	✓	✓	✓	✓		✓
StickyTags [88]	MS	Spatial	8-9	1 4-5						
LowFat [69]	MS	Spatial	3	1 4-5						
EffectiveSan [70]	MS	Spatial, Temporal, Miscasts	3 10-13	1						
EffectiveSan-bounds [70]	MS	Spatial	3	1 4-5						
EffectiveSan-type [70]	MS	Miscasts	10-11	1 3 5						
HexType [110]	MS	Miscasts	4	1 3 5						
PACMem [136]	MS	Spatial, Temporal	8-9 12-13	1 4						
CETS [160]	MS	Temporal	12-13 8 5	1 3-4						
DangSan [219]	MS	Temporal	5	1 3-4						
MarkUs [3]	MS	Temporal	5	1 3-4						
FFMalloc [231]	MS	Temporal	1-2	1 3-4						
SafeInit [157]	MS	Uninitialized	1	3-5						
Chow et al. [48]	MS	Uninitialized	1	3-5						
CHERI [233]	MS, IoD, IoU	Spatial	8-9 1-6 8 5 7 3-4	1 4-5						
Cornucopia [79]	MS	Temporal	5	1 4 8-9 1-6 8 5 7 3-4						
Mon CHÉRI [94]	MS	Uninitialized	7	4-5 8-9 1-6 8 5 7 3-4						
$W \oplus X$ [240]	IoD,IoU	-	3-4 4		✓	✓	✓	✓	✓	✓
WIT [4]	IoD, IoU	DFI	2-3 6 X 3		✓	✓	✓	✓	✓	✓
DSR [23, 34]	IoD	Data Randomization			✓	✓	✓	✓	✓	✓
DFI [38]	IoD, IoU	DFI	4 1-9 2-4 3-4 8							
SafeStack [126]	IoD	CPI	5 12	7 10 4 6 11	✓	✓	✓			
CPS [126]	IoD	CPI	5-6 12 7	4	✓	✓	✓	✓		
CPI [126]	IoD	CPI	5-6 11-12 7 10	4	✓	✓	✓			
IFCC [214]	IoU	CFI	2-3 6 3	4-5 12	✓	✓	✓			
Intel IBT [215]	IoU	CFI	2 3	4-5 12	✓	✓	✓	✓		
FineIBT [80]	IoU	CFI	2-3 6 3	4-5 12	✓	✓	✓			
Intel Shadow Stack [215]	IoU	CFI	1 4-5 4	7 10 4 6 11	✓	✓	✓			
Intel CET [215]	IoU	CFI	1-2 4-5 3-4	4	✓	✓	✓	✓		
eXec-Only Memory [244]	IoU	XOM	4 1		✓	✓	✓	✓	✓	✓
ILR [98]	IoD	Code Randomization	**CLK, CCK**	3-4	✓	✓	✓	✓	✓	✓
ILR+XOM [98, 244]	IoD	Code Randomization	4 1 **CLK, CCK**	3-4	✓	✓	✓	✓		
Readactor++ [58]	IoD	Code Randomization	4 1 2 **CLK, CCK**	3-4	✓	✓	✓	✓		
Cox et al. (MVX+ASP) [56]	IoU	MVX	2 4 1-4 3-4 6 5 **CLK**		✓	✓	✓	✓		
ReMon (MVX+DCL) [223, 224]	IoU	MVX	2 1-4 **CLK**	3-4	✓	✓	✓	✓		
MvArmor (MVX+ASP-DLK) [124]	IoD, IoU	MVX	1-4 1-4 3-4 8 5 **DLK, CLK**							
KENALI [205]	IoD	Sensitive Data Integrity	7 9 6 9	4-6 11-12 7 10	✓					
TRUVIN [82]	IoU	Sensitive Data Integrity	7 8	4-6 11-12 7 10	✓					

Legend: **CF:** Control Flow, **ACE:** Arbitrary Code Execution, **MS:** Memory Safety, **IoD:** Isolation of Data, **IoU:** Integrity of Use, **CLK:** Code Layout Knowledge, **CCK:** Code Content Knowledge, **DLK:** Data Layout Knowledge, **MVX:** Multi-Variant eXecution, **CPI:** Code Pointer Integrity, **CFI:** Control Flow Integrity, **DFI:** Data Flow Integrity, ✓ Attacker has achieved this goal.

defenses[1] onto the graph, many of which are deployed in practice or widely studied, as shown in Table 4; an extended mapping of additional defenses is provided in Table 6 in Appendix A. Our model is implemented using a custom graph structure in Python, and we implement each defense mapping as a small, declarative method that marks certain edges as inhibited on the graph. Listing 1 illustrates the mapping for Safe Rust: it inhibits edges that would create invalid pointers (out-of-bounds, dangling, or type-confused) from a valid data-pointer state and prevents the creation of references to uninitialized memory. Additional details on using our model, along with further examples, are provided in Appendix B.

Together, the selected defenses in Table 4 cover 98% of the defensible edge space. The only edge never directly inhibited is 1, which models direct execution of attacker-corrupted code memory. While this could technically be inhibited (IoU-style), modern operating systems instead address the code corruption attack vector IoD-style by enforcing read-only code as part of the W⊕X policy [240], which many existing defenses assume is already present.

For each defense, we model not only the edges it explicitly blocks but also the edges it assumes are already mitigated as part of its threat model. This distinction exposes how defenses are intended to compose with others, e.g., FineIBT [80], like all CFI schemes, assumes some baseline mitigations, such as W⊕X [240].

We then run an automated fragility test: an iterative reachability analysis over the entire graph. Starting only from the

[1]We include some "debugging" sanitizer designs, modeled as runtime exploit mitigations. This is common practice [207].

```
@prettyname("Safe Rust")
def Rust(graph: ExploitGraph, explorer: Explorer):
    explorer.inhibit(
        graph.find_edge('v_dp_reg', 'dp_to_free'),
        graph.find_edge('v_dp_reg', '
            dp_to_type_confused'),
        graph.find_edge('v_dp_reg', 'dp_to_oob'),
        graph.find_edge('alloc', 'dp_to_uninit'),
    )
```

Listing 1: Python model mapping for Safe Rust [151].

payload and **Allocated memory** abilities, we derive all subsequent capabilities, by enabling nodes whose incoming edges are fully satisfied. To structure the results in Table 4, we distinguish high-level semantic **Attacker Goals** (Information Disclosure, Access Control, Arbitrary Computation) from low-level **Execution Control** techniques (Whole-Function Reuse, Instruction-Granular control-flow hijack, Arbitrary Code Execution). A defense, under its stated threat model, is considered *design-level fragile* if any attack goal node remains reachable.

On Edge Fragility. Some defenses apply coarse-grained policies when protecting exploitable edges in the graph. Although such defenses may inhibit the same edges as their fine-grained counterparts at the design level, they can exhibit what we refer to as *edge fragility* when these edges remain exploitable in practice. This fragility can arise from imprecise enforcement boundaries, such as unprotected sub-object overflows [85], equivalence-class imprecision [35, 75, 146], or limited redzone coverage [88], which allow specific exploit conditions to bypass the intended protection.

We consider edge fragility to be distinct from the design-level fragility analyzed in this work. Edge fragility conditions depend on concrete execution instances and granularity levels and, therefore, require empirical validation, which we leave to future work. Nevertheless, the model could still serve as a centralized repository to collect known edge-fragility cases (however defense-specific) on specific edges, and warn users before marking them inhibited.

5 Systemization of Defense Fragility

With defenses mapped onto the graph (Section 4) and the corresponding fragility tests executed, we can systematically assess whether each defense's stated protection goals hold under its assumed threat model. We summarize the main findings and the design-level fragility patterns that emerge.

For the first-level exploit mitigations (top half of Table 4), this analysis reveals no trivial design-level fragility that would completely undermine their primary protection goals. **Rust** [151] and **CCured** [163] achieve this without relying on external assumptions. The only exception is **SoftBound's store-only mode** [159]: when read checks are disabled, our model automatically highlights paths for all four store-only bypass attacks listed by Jacobs et al. [109].

Among first-level defenses, protections against use-after-free (UAF) vulnerabilities are the most diverse. **EffectiveSan** [70], **PACMem** [136], and **CETS** [160] implement the classical Lock-and-Key approach, which detects temporal violations during dereferences. In contrast, **DangSan** [219] and similar techniques [130, 142, 241] actively eliminate all references to freed memory upon deallocation, while **MarkUs** [3] passively waits for dangling pointers to disappear before reusing memory. **FFMalloc** [231] takes a more radical approach by never reusing virtual addresses. Our model shows that this no-reuse policy not only prevents UAF exploitation but also eliminates uninitialized read vulnerabilities on the heap.

CHERI [230] capabilities can natively enforce spatial memory safety and referential integrity: capabilities carry bounds and are tagged using a disjoint validity bit, preventing both capability forging and capability overwrites. Capability substitution attacks are not natively inhibited. CHERI does use different capabilities for code and data, which blocks code overwrites (using data capabilities) and data execution (using code capabilities) within a compartment. We additionally model **Cornucopia** [79] and **Mon CHÉRI** [94] to capture temporal safety and uninitialized read protection on CHERI platforms. If type miscast bugs are not explicitly ruled out, however, both our model and recent analyses [114, 216] highlight capability substitution attacks, where existing valid capabilities are misused via type-confused reads or writes. These attacks do not break compartmentalization, but they can still enable coarse-grained control-flow hijacking within a CHERI application.

Takeaway-2: Early-stage defenses tend to be resilient under their stated assumptions, but can be fragile under relaxed variants, such as disabling read checks. They also rarely cover all initial vulnerability types.

WIT [4] is an example of a defense that spans strategies. It statically derives a precise set of valid write targets for each store instruction, thereby enforcing Isolation of Data (IoD) for all program data. It also provides Integrity of Use (IoU) for data pointers: even if a pointer is corrupted, it cannot be used to write beyond its statically allowed targets. Beyond write protection, WIT includes a fine-grained forward-edge CFI—long considered one of the most precise implementations to date [181]. With this design, return addresses are never valid targets, and function pointer hijacks are blocked by IoU checks, making WIT highly effective against control-flow hijacks according to our fragility test. Aside from information leakage, the absence of checks on reads leaves room for data-only attacks through crafting and substitution.

Our model also exposes the design-level fragility difference

between **CPI** and **CPS** [126]. CPI recursively protects not only code pointers but also pointers to these pointers, essentially enforcing SoftBound-style [159] safety on all operations that directly or indirectly handle code pointers. This prevents overwrites, crafting, and substitution attacks. CPS, in contrast, protects only direct overwrites, leaving substitution attacks possible and still allowing coarse-grained WFR. Both variants rely on **SafeStack** [126] for backward-edge protection.

Takeaway-3: Precise write-side restrictions and recursive pointer protection can limit control-flow hijacks, but their guarantees can become fragile when residual read surfaces enable pointer crafting or substitution, leaving room for data-only and coarse-grained code reuse attacks.

ILR [98] represent a class of fine-grained code randomization defenses [99, 177] that assume code-reuse attacks become infeasible without gadget-location knowledge. Our automated analysis rejects this hypothesis (as shown in Table 4) in the presence of information disclosure: JIT-ROP–style attacks [204] can leak code content, reconstruct gadget sets offline, and then launch control-flow hijacks. Adding eXecute-Only Memory [15, 212, 244] to block code reads (**ILR+XOM** [98, 244]) addresses this particular leak but not the more fundamental issue that data memory often holds function and return-site pointers [57]. Our model confirms that leaking these pointers still enables coarse-grained control-flow hijacking.

Defenses like **Readactor++** [57, 58] introduce Code Pointer Hiding by replacing code pointers with opaque identifiers. Table 4 shows the fragility of this approach, matching AOCR's findings [189]: even if code addresses are hidden behind identifiers, these identifiers can be reused and *substituted* as proxies for code pointers, permitting coarse-grained code reuse. AOCR thus argues for the inherent limits of code randomization alone against code reuse, pushing research toward data randomization [21]. To verify the claim, we model a hypothetical **Readactor++** [58] variant augmented with data layout randomization (no-DLK). Our model then identifies data address leakage as a threat to layout secrecy. If we additionally prevent such leakage, the model confirms that both code-reuse and data-only attacks are effectively mitigated. Indeed, this aligns with recent AOCR-resilient proposals [21].

Takeaway-4: Hiding and code randomization are fragile once information disclosure or code reuse channels exist: blocking code reads helps, but leaked pointers (or pointer proxies) can still re-enable coarse-grained code reuse.

Multi-Variant Execution (MVX) systems [56] prevent some forms of information leakage by running diversified variants in parallel and checking for divergences. Cox et al. [56] showed that variants with non-overlapping address spaces (**MVX+ASP** [56] in Table 4) deterministically block absolute-address leakage and injection. Later, Volckaert et al. [223] demonstrated that diversifying code layouts alone suffices to prevent code-reuse attacks that rely on injected code addresses (**MVX+DCL** [223, 224]). Our model confirms these results for attacks that externalize code pointers, i.e., for **READ** Leak and **WRITE** Overwrite. However, it also confirms Göktaş et al.'s observation [93] that reuse of in-program code pointers (**READ** Substitute and **WRITE** Substitute) can bypass such protections and re-enable fine-grained code reuse attacks. Therefore, recent MVX work incorporates data layout randomization [192]. **MvArmor** [124] applies fine-grained data layout diversification and, in our analysis, eliminates all viable attack paths when both code and data are diversified. As summarized in Table 4, **MvArmor** [124] is one of a small set of defenses, including **DFI** [38], **DSR** [23, 34], **Rust** [151], and **CCured** [163] that mitigate all modeled attack vectors without relying on external protections such as DEP [154].

Finally, we model two defenses that explicitly target data-only attacks while ruling out control-flow hijacking and leakage attacks. **TRUVIN** [82] identifies security-critical variables whose corruption could enable privilege escalation [39] or the construction of gadget-dispatcher loops [101] and protects them through IoU-style checks. Likewise, **KENALI** [205] identifies critical non-control data in the Linux kernel and protects them via IoD. Under their stated threat models, our model confirms that data-only attacks are effectively mitigated.

Takeaway-5: Diversification is least fragile when it simultaneously removes both code and data reuse channels, whereas defenses targeting a specific attacker goal (e.g., data-only attacks) can be robust within that scope because they explicitly constrain what they promise to prevent.

6 Fragility Testing: Use Cases

Building on our model of potential attack vectors (Section 3) and the corresponding mapping of representative defenses (Section 4), our infrastructure enables automated reasoning about the security of defenses under hypothetical deployment scenarios. In this section, we use our model to study the interoperability between defenses in multi-language applications, and to assess the portability of their security guarantees across different deployment contexts.

6.1 Interoperability Testing

6.1.1 Motivation and Objectives

Recent work has shown that incrementally migrating security-hardened C/C++ codebases to memory-safe languages such as Rust can reintroduce previously mitigated attack vectors [152, 176]. For instance, Mergendahl et al. [152] demonstrated that control-flow hijacking becomes possible when a CFI-hardened C component corrupts a code pointer that

is later invoked by Safe Rust. Although CFI and Rust independently defend against such attacks, their guarantees fail to compose due to mismatched assumptions at the language boundary. The Rust compiler community has since acknowledged the need to support memory safety mitigations, such as CFI, in Safe Rust code as well [89], in order to avoid making the transition to Rust an all-or-nothing effort.

These so-called cross-language attacks (CLAs) are not specific to Rust and have been demonstrated across multiple language or defense combinations [31, 66, 134, 156, 176]. The key observation is that defenses operating at *different stages* of the exploit lifecycle *implicitly assume different attacker capabilities*. When attackers can split up exploit steps across different components, latent attack paths may re-emerge.

Our model generalizes this observation by enabling systematic and automatic [2] *interoperability* testing between arbitrary sets of defenses, assuming two interacting application components. Due to space constraints, we focus on the specific scenario of memory-unsafe components in otherwise Safe Rust programs in this paper. We first examine whether interoperability issues arise in our model when combining CFI-hardened C code with Safe Rust. We then evaluate the hypothesis of Mergendahl et al. [152] that, in this setting, CLAs can be prevented by pairing Safe Rust with C side defenses that stop pointer invalidation at the earliest stage of exploitation. Accordingly, we restrict our analysis to the first level exploit mitigations illustrated in Table 4, namely those employing a memory safety (MS) strategy, and systematically analyze their interaction with Safe Rust in our model.

6.1.2 Interoperability Semantics and Validation

To evaluate the interoperability of two defenses, our model first determines the set of attacker capabilities that remain reachable under each defense in isolation. It then computes the union of these residual capabilities and re-evaluates each defense under this combined capability set. This analysis assumes that attacker capabilities can freely transfer across components, i.e., there is no additional isolation or validation between them. This matches Mergendahl et al.'s threat model [152]. Enforcing better isolation is the focus of ongoing work on CLA defenses [18, 116, 122], but remains a non-trivial challenge [116].

We begin by evaluating interoperability between Safe Rust and FineIBT [80] combined with Intel's Shadow Stack [215], in order to validate our model's ability to anticipate impedance mismatches. We choose this combination because FineIBT together with Shadow Stack represents one of the strongest forms of CFI deployed in practice, providing fine-grained protection for both forward and backward control-flow edges. While our automated fragility test shows that each defense in isolation (Safe Rust vs FineIBT + Shadow Stack) eliminates control-flow hijacking attack vectors (see Table 4), the interoperability test confirms the re-emergence of a potential control-flow hijacking path. Specifically, our model identifies a path enabling both coarse-grained and fine-grained control-flow hijacking when attacker capabilities are transferred across components.

[2] Only the defense mapping requires manual effort; all subsequent tests are fully automated.

Table 5: First-level defenses that interact well with Rust.

Category	Exploit Mitigation	Safe Rust [151]
Spatial	LowFat [67] (object-based)	Static pointer arithmetic restrictions (object-based)
Temporal	MarkUs [3] (GC) DangSan [219] (dangling pointer revocation)	Compile-time restrictions, reference counting (GC)
Initialization	Chow et al. [48] (zero on `free`) SafeInit [157] (init on `alloc`)	Auto-initialization on `alloc`
Type	HexType [110] (cast verification)	Type casting verification

Note that the individually mentioned defenses merely represent a unique defensive approach, and could each be substituted for a different defense with the same design-level characteristics but different implementation choices. By no means is this the exhaustive list of defenses that securely interact with Safe Rust code.

Next, we consider the **14** candidate first-level defenses (underlined in Table 4) selected for interoperability testing. Since only CCured [163] and EffectiveSan [70] cover multiple classes of memory safety violations in their design, we use our model to automatically enumerate all minimal combinations of first-level defenses that collectively block every attacker goal. The model identifies **55** such combinations. Each combination is *complete* (blocking all pointer invalidation paths) and *minimal* (removing any defense re-enables at least one attacker goal).

We then evaluate the interoperability of each combination with Safe Rust by transferring the residual attacker capabilities enabled by the C/C++ defenses into the Rust component. Our model concludes that only **4** combinations, comprising **6** distinct defenses, eliminate all CLA paths. These defenses, summarized in Table 5, share a common design principle: *they prevent pointers from becoming invalid in the first place*. In contrast, defenses that detect pointer invalidity at dereference time compose poorly with memory-safe languages, as they implicitly rely on *Integrity-of-Use*-style checks within their memory safety strategy. Examples include pointer-based bounds checking defenses such as SoftBound [159] and CCured [163], lock-and-key temporal safety mechanisms such as PACMem [136] and CETS [160], as well as late type-verification approaches such as EffectiveSan [70]. While effective in isolation, these mechanisms permit invalid pointers to exist transiently, violating Rust's core assumption that all incoming pointers are memory-safe. As a result, they enable CLAs unless additional sanitization or isolation is enforced at the language boundary.

Takeaway-6: Interoperation can introduce design-level fragility. Defenses that follow different strategies (Section 2.2) may fail to compose when deployed together (e.g., CFI (`Integrity-of-Use`) and Safe Rust (`Memory Safety`)). Importantly, interoperability failures can also arise between defenses that nominally follow the same strategy (e.g., CCured and Safe Rust). Ensuring interoperability among defenses is non-trivial and requires explicit, careful consideration when defenses are combined across deployment scenarios, particularly in polyglot and heterogeneous applications.

6.2 Portability of Security Guarantees

As memory-unsafe software increasingly extends beyond CPUs, memory errors do so as well [90]. Recent work shows that many classical CPU exploitation techniques and memory corruption primitives readily carry over to GPUs [64, 90, 155, 186], which today constitute the dominant accelerator platform for Machine Learning workloads [186]. Worse yet, many of these platforms lack even basic memory protection features such as DEP or read-only code pages, let alone more advanced memory tagging or control-flow enforcement features on which modern defenses depend [199].

Our model enables automated validation of the *portability* of security guarantees across such platforms. To assess the fragility of the representative defenses in Table 4 when deployed on GPUs, we re-run the same fragility analysis under a threat model that excludes DEP and read-only code (i.e., no W⊕X), reflecting the protection mechanisms currently absent on modern GPUs [90].

Under this setting, our analysis shows that none of the first-level exploit mitigations are affected, nor are Data Flow Integrity (DFI) [38] or Data Space Randomization (DSR) [23] mechanisms. These defenses already provide sufficient memory-safety guarantees to prevent payload injection and code corruption, and therefore do not fundamentally rely on an active W⊕X policy in their threat models.

In contrast, our model reveals that all control-flow hijacking mitigations (e.g., Code Pointer Integrity (CPI) [126] and Control Flow Integrity (CFI) [1]) become fragile under these conditions, primarily due to the absence of code overwrite protection (4). These defenses assume that arbitrary code execution can only be achieved via control-flow hijacking and therefore focus exclusively on protecting code pointers. However, on platforms, e.g., GPUs, where code pages are writable, adversaries can inject and execute code directly without corrupting code pointers, bypassing these defenses.

Finally, our model reveals that Multi-Variant Execution (MVX) systems exhibit differing levels of resilience to the absence of DEP. With fine-grained data-layout randomization applied across variants [124], security guarantees remain intact, as injected payloads reside at unpredictable locations in each variant. Under disjoint address spaces [56], payload addresses are predictable but differ across variants, still preventing consistent shellcode execution. In contrast, when only code layouts differ across variants [223], injected payloads remain consistently executable and MVX diversification provides no protection. Consequently, only Disjoint Code-Layout MVX designs rely on DEP in their threat models [223].

Takeaway-7: Security guarantees are not inherently portable across platforms. Defenses that implicitly rely on platform-provided memory protections, such as DEP or non-writable code, might become fragile when deployed on architectures where these assumptions do not hold. In contrast, defenses that rely solely on platform-agnostic memory-safety strategies remain robust across platforms, highlighting *portability* as an underexplored dimension of design-level fragility.

7 Related Systemizations

Several systematization efforts have examined distinct aspects of the memory safety domain. Song et al. [207] provide a systematic overview of sanitizers (generally denoted "Memory Safety Enforcement" in Section 2.2.1), focusing primarily on design or implementation aspects, and their effectiveness in detecting vulnerabilities. Larsen et al. [128] present a comprehensive analysis of software diversification techniques, a class of probabilistic exploit mitigations of which we described part of the fragility history in Section 5. Burow et al. [33] explore the design space of shadow stacks, thoroughly evaluating their performance, compatibility, and security implications. Ammar et al. [9] present a systematization of Control Flow Integrity (CFI) and Control Flow Attestation (CFA) mechanisms where they examine the differences and similarities between these defense classes. Lefeuvre et al. [132] recently systematized the field of software compartmentalization, which deals with containing the fallout from software vulnerabilities in general, including memory errors.

Finally, the closest work to our systematization is the seminal SoK by Szekeres et al. [211], which remains one of the most comprehensive efforts to systematize the landscape of memory corruption defenses. It already proposed a high-level graph-based overview of possible attacks, and breaks down exploitation into five distinct stages on which different defense categories map. Szekeres et al.'s work has been instrumental in advancing the community's understanding of exploit mitigations and their practical trade-offs in terms of performance, compatibility, and robustness, in particular as they relate to the chances of adoption. In contrast, our SoK serves a different purpose: to specifically model the *security* guarantees of defenses, and to make explicit all of the design assumptions behind the intended security properties. This requires a more systematic and expressive attack model, which is what

we aimed to provide through our detailed graph structure, as discussed in the previous sections.

8 Looking Back, Looking Ahead!

After three decades, the battle for memory safety has been waged with remarkable creativity and determination, yet the war is far from over. Attacks that play on the rope of (design-level) defense *fragility* do not emerge from flawed prototype implementations, but rather from flawed assumptions at design time. In the age of generative AI, heterogeneous computing, and polyglot software stacks, manual, ad hoc reasoning about security guarantees is no longer sufficient. Therefore, there is a need to anticipate fragility, not just react to it. The framework proposed in this paper is a small step; a lens to view the hurdles and opportunities ahead, especially with emerging computing architectures, such as GPUs and NPUs. The next frontier of memory safety is not merely faster mitigation but stronger guarantees. Achieving this demands a fundamental shift from *fragile* evaluation practices to *systematized* evaluation methodologies that make assumptions explicit and, when feasible, enable automated fragility testing.

Open Science

In support of open science, we publicly release the tooling that implements our graph-based exploitation model and the associated defense-mapping infrastructure.

The released artifacts include (1) the formal graph model encoding memory corruption capabilities and transitions, (2) declarative mappings of defenses to the edges they inhibit and the assumptions they rely on, and (3) analysis scripts used to perform automated reachability analyses for fragility and interoperability testing. Together, these artifacts enable researchers and practitioners to reproduce the analyses presented in this paper, evaluate additional defenses, and explore new deployment scenarios and threat models.

Our artifacts are preserved on Zenodo at https://doi.org/10.5281/zenodo.20315524, and on GitHub at https://github.com/adriaanjacobs/sec26-sok-fragility.

Ethical Considerations

This work is primarily analytical and systematizing in nature. It does not introduce new exploitation techniques, vulnerabilities, or attack implementations. Instead, it systematizes existing knowledge on previously published attack primitives and defense mechanisms to reason about their security guarantees and fragility under varying assumptions.

By making defense assumptions explicit and highlighting design-level gaps, this work may expose limitations in existing mitigations. We believe that such transparency ultimately benefits defenders by enabling more robust designs, better-informed deployment decisions, and earlier identification of fragile security assumptions. Our framework is intended to support defensive evaluation and does not lower the barrier for real-world exploitation beyond what is already described in the existing literature.

The released tooling operates at an abstract modeling level and does not generate exploits. Rather, it evaluates the robustness of defenses at the design level under varying adversarial assumptions and deployment scenarios. By enabling systematic reasoning about mitigation fragility, this work aims to reduce reliance on ad hoc security arguments and contribute to the development of more robust memory safety defenses.

Acknowledgments

We thank the anonymous reviewers for their feedback. We also thank Dimitris Stavrakakis, Márton Bognár, and Frank Piessens for their feedback on earlier drafts of this paper. This research is partially funded by the Internal Funds KU Leuven, and by the Cybersecurity Research Program Flanders.

References

[1] Abadi, M., et al. Control-flow Integrity Principles, Implementations, and Applications. *ACM TISSEC* (2009).

[2] Ahmed, S., et al. Not All Data are Created Equal: Data and Pointer Prioritization for Scalable Protection Against Data-Oriented Attacks. In *Proc. USENIX Security* (2023).

[3] Ainsworth, S., and Jones, T. M. MarkUs: Drop-in use-after-free prevention for low-level languages. In *SP* (2020).

[4] Akritidis, P., et al. Preventing Memory Error Exploits with WIT. In *IEEE S&P* (2008).

[5] Akritidis, P., et al. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense Against Out-of-Bounds Errors. In *USENIX Security* (2009).

[6] Almakhdhub, N. S., et al. μRAI: Securing Embedded Systems with Return Address Integrity. In *NDSS* (2020).

[7] Ammar, M., et al. SIMPLE: A Remote Attestation Approach for Resource-constrained IoT Devices. In *ICCPS* (2020), IEEE.

[8] Ammar, M., et al. On Bridging the Gap between Control Flow Integrity and Attestation Schemes. In *USENIX Security* (2024).

[9] Ammar, M., et al. SoK: Integrity, Attestation, and Auditing of Program Execution. In *IEEE S&P* (2025).

[10] Apple. Operating System Integrity. https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b/web, 2021. [Online; accessed 18-February-2023].

[11] ARM. Return Address Signing using ARM Pointer Authentication. https://gcc.gnu.org/legacy-ml/gcc-patches/2018-11/msg00104.html, 2018. [Online; accessed 13-February-2023].

[12] ARM. BTI. https://developer.arm.com/documentation/ddi0602/2021-12/Base-Instructions/BTI--Branch-Target-Identification-, 2020. [Online; accessed 13-February-2023].

[13] ARM. Learn the Architecture - Providing Protection for Complex Software. https://developer.arm.com/documentation/102433/0100, 2022. [Online; accessed 18-February-2023].

[14] AZEVEDO DE AMORIM, A., ET AL. The meaning of memory safety. In *POST* (2018).

[15] BACKES, M., ET AL. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM CCS* (2014).

[16] BACKES, M., AND NÜRNBERGER, S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security* (2014).

[17] BADOUX, N., ET AL. Type++: Prohibiting Type Confusion with Inline Type Information. In *NDSS* (2025).

[18] BANG, I., ET AL. TRust: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code. In *Proc. USENIX Security* (2023).

[19] BELLEVILLE, B., ET AL. KALD: Detecting Direct Pointer Disclosure Vulnerabilities. *IEEE Trans. Dependable Secure Comput.* (2021).

[20] BERGER, E. D., AND ZORN, B. G. DieHard: probabilistic memory safety for unsafe languages. In *ACM PLDI* (2006).

[21] BERLAKOVICH, F., AND BRUNTHALER, S. R2C: AOCR-Resilient Diversity with Reactive and Reflective Camouflage. In *Proceedings of the Eighteenth European Conference on Computer Systems* (2023), EuroSys '23.

[22] BHATKAR, S., ET AL. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *USENIX Security* (2005).

[23] BHATKAR, S., AND SEKAR, R. Data Space Randomization. In *DIMVA* (2008).

[24] BIERBAUMER, B., ET AL. Smashing the stack protector for fun and profit. In *IFIP SEC* (2018), Springer.

[25] BIGELOW, D., ET AL. Timely Rerandomization for Mitigating Memory Disclosures. In *CCS* (2015).

[26] BINOSI, L., ET AL. The Illusion of Randomness: An Empirical Analysis of Address Space Layout Randomization Implementations. In *ACM CCS* (2024).

[27] BISWAS, P., ET AL. Venerable Variadic Vulnerabilities Vanquished. In *USENIX Security* (2017).

[28] BITTAU, A., ET AL. Hacking blind. In *IEEE S&P* (2014).

[29] BLETSCH, T., ET AL. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS* (2011).

[30] BOEHM, H.-J., AND WEISER, M. Garbage collection in an uncooperative environment. *Softw. Pract. Exp.* (1988).

[31] BROWN, F., ET AL. Finding and preventing bugs in javascript bindings. In *IEEE Symposium on Security and Privacy (SP)* (2017).

[32] BUROW, N., ET AL. Control-flow integrity: Precision, security, and performance. *ACM CSUR* (2017).

[33] BUROW, N., ET AL. SoK: Shining Light on Shadow Stacks. In *IEEE S&P* (2019).

[34] CADAR, C., ET AL. Data Randomization. Tech. rep., Technical Report TR-2008-120, Microsoft Research, 2008.

[35] CARLINI, N., ET AL. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security* (2015).

[36] CARLINI, N., AND WAGNER, D. ROP is still dangerous: Breaking modern defenses. In *USENIX Security* (2014).

[37] CARR, S. A., AND PAYER, M. Datashield: Configurable data confidentiality and integrity. In *AsiaCCS* (2017).

[38] CASTRO, M., ET AL. Securing Software by Enforcing Data-Flow Integrity. In *OSDI* (2006).

[39] CHEN, S., ET AL. Non-control-data attacks are realistic threats. In *USENIX Security* (2005).

[40] CHEN, W., ET AL. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *USENIX Security* (2020).

[41] CHEN, X., ET AL. Limitations and opportunities of modern hardware isolation mechanisms. In *USENIX ATC* (2024).

[42] CHEN, Y. Advanced exploit techniques attacking the ie script engine. https://herrymorison.tistory.com/entry/Advanced-Exploit-Techniques-Attacking-the-IE-Script-Engine.

[43] CHEN, Y. Advanced Exploit Techniques Attacking the IE Script Engine. https://herrymorison.tistory.com/entry/Advanced-Exploit-Techniques-Attacking-the-IE-Script-Engine.

[44] CHEN, Y.-C., AND LI, S.-W. HeMate: Enhancing Heap Security through Isolating Primitive Types with Arm Memory Tagging Extension. In *ARES* (2024).

[45] CHENG, L., ET AL. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM TOPS* (2021).

[46] CHISNALL, D., ET AL. CHERI JNI: Sinking the Java Security Model into the C. In *Proc. 22nd Int. Conf. Archit. Support Prog. Lang. OS (ASPLOS)* (2017).

[47] CHO, H., ET AL. ViK: practical mitigation of temporal memory safety violations through object ID inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022), ASPLOS '22.

[48] CHOW, J., ET AL. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *USENIX Security* (2005).

[49] CHRISTOULAKIS, N., ET AL. HCFI: Hardware-enforced control-flow integrity. In *CODASPY* (2016).

[50] CISA. Exploring Memory Safety in Critical Open Source Projects. Tech. rep., Cybersecurity and Infrastructure Security Agency, 2024.

[51] CONTI, M., ET AL. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *CCS* (2015).

[52] COOK, K. Introduce struct layout randomization plugin. https://lwn.net/Articles/723997/, 2017.

[53] COPPENS, B., ET AL. Multi-variant execution environments. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. 2018.

[54] CORPORATION, T. M. CWE-457: Use of Uninitialized Variable. https://cwe.mitre.org/data/definitions/457.html, 2021.

[55] CORPORATION, T. M. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html, 2024.

[56] COX, B., ET AL. N-Variant Systems: A Secretless Framework for Security through Diversity. In *USENIX Security* (2006).

[57] CRANE, S., ET AL. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE S&P* (2015).

[58] CRANE, S. J., ET AL. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)* (2015).

[59] DANG, T. H., ET AL. Oscar: A practical {Page-Permissions-Based} scheme for thwarting dangling pointers. In *26th USENIX Security Symp. (USENIX Security)* (2017).

[60] DAVI, L., ET AL. Stitching the gadgets: On the ineffectiveness of Coarse-Grained Control-Flow integrity protection. In *USENIX Security* (2014).

[61] DAVI, L., ET AL. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *NDSS* (2015).

[62] DAVI, L. V., ET AL. Gadge Me If You Can: Secure and Efficient Ad-Hoc Instruction-Level Randomization for x86 and ARM. In *ACM CCS* (2013).

[63] DHAR, A., ET AL. Ascend-CC: Confidential Computing on Heterogeneous NPU for Emerging Generative AI Workloads. *arXiv preprint arXiv:2407.11888* (2024).

[64] DI, B., SUN, J., AND CHEN, H. A study of overflow vulnerabilities on GPUs. In *Network and Parallel Computing: 13th IFIP WG 10.3 International Conference, NPC 2016, Xi'an, China, October 28-29, 2016, Proceedings 13* (2016), Springer, pp. 103–115.

[65] DING, R., ET AL. Efficient Protection of Path-Sensitive Control Security. In *USENIX Security* (2017).

[66] DRAKE, J. Exploiting memory corruption vulnerabilities in the java runtime. *Black Hat Abu Dhab* (2011).

[67] DUCK, G. J., ET AL. Stack Bounds Protection with Low Fat Pointers. In *NDSS* (2017).

[68] DUCK, G. J., ET AL. Hardening Binaries Against More Memory Errors. In *EuroSys* (2022).

[69] DUCK, G. J., AND YAP, R. H. Heap Bounds Protection with Low Fat Pointers. In *CC* (2016).

[70] DUCK, G. J., AND YAP, R. H. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. In *PLDI* (2018).

[71] DULLIEN, T. Weird machines, exploitability, and provable unexploitable. *IEEE Trans. Emerg. Top. Comput.* (2017).

[72] DURUMERIC, Z., ET AL. The matter of heartbleed. In *IMC* (2014).

[73] DUTA, V., ET AL. Let Me Unwind That For You: Exceptions to Backward-Edge Protection. In *NDSS* (2023).

[74] ELLIOTT, A. S., ET AL. Checked C: Making C Safe by Extension. In *IEEE SecDev* (2018).

[75] EVANS, I., ET AL. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM CCS* (2015).

[76] EVANS, I., ET AL. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *IEEE S&P* (2015).

[77] FARKHANI, R. M., ET AL. On the effectiveness of type-based control flow integrity. In *ACSAC* (2018).

[78] FARKHANI, R. M., ET AL. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *USENIX Security* (2021).

[79] FILARDO, N. W., ET AL. Cornucopia: Temporal safety for CHERI heaps. In *IEEE Symp. Security Privacy (SP)* (2020).

[80] GAIDIS, A. J., ET AL. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In *RAID* (2023).

[81] GE, X., ET AL. Griffin: Guarding control flows using intel processor trace. *ACM SIGPLAN Notices* (2017).

[82] GEDEN, M., AND RASMUSSEN, K. TRUVIN: Lightweight Detection of Data Oriented Attacks Through Trusted Value Integrity. In *TrustCom* (2020).

[83] GEDEN, M., AND RASMUSSEN, K. RegGuard: Leveraging CPU Registers for Mitigation of Control- and Data-Oriented Attacks. *Computers & Security* (2023).

[84] GEORGE, R., ET AL. OPTISAN: Using Multiple Spatial Error Defenses to Optimize Stack Memory Protection within a Budget. In *USENIX Security* (2024).

[85] GIL, R., ET AL. There's a hole in the bottom of the c: On the effectiveness of allocation protection. In *2018 IEEE Cybersecurity Development (SecDev)* (2018).

[86] GODEFROID, P. Fuzzing: Hack, Art, and Science. *Communications of the ACM* (2020).

[87] GOOGLE. Retrofitting Spatial Safety to Hundreds of Millions of Lines of C++, 2024.

[88] GORTER, F., ET AL. Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags. In *IEEE S&P* (2024).

[89] GROUP, R. E. M. P. Rust Exploit Mitigations PG Roadmap for 2023. https://hackmd.io/@rcvalle/H1epy5Xqn, 2023.

[90] GUO, Y., ET AL. GPU Memory Exploitation for Fun and Profit. In *USENIX Security* (2024).

[91] GÖKTAS, E., ET AL. Out of control: Overcoming control-flow integrity. In *IEEE S&P* (2014).

[92] GÖKTAS, E., ET AL. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *USENIX Security* (2014).

[93] GÖKTAS, E., ET AL. Position-independent code reuse: On the effectiveness of ASLR in the absence of information disclosure. In *EuroS&P* (2018), IEEE.

[94] GÜLMEZ, M., ET AL. Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities. In *IEEE S&P* (2025).

[95] HAGER-CLUKAS, A., AND HOHENTANNER, K. DMTI: Accelerating Memory Error Detection in Precompiled C/C++ Binaries with ARM Memory Tagging Extension. In *ACM ASIACCS* (2024).

[96] HALLER, I., ET AL. TypeSan: Practical Type Confusion Detection. In *ACM CCS* (2016).

[97] HASSAN, M. T. B. M. *Hardware-Software Co-design for Practical Memory Safety*. PhD thesis, Columbia Univ., USA, 2022.

[98] HISER, J., ET AL. ILR: Where'd My Gadgets Go? In *IEEE S&P* (2012).

[99] HOMESCU, A., ET AL. Large-Scale Automated Software Diversity—Program Evolution Redux. *IEEE Trans. Dependable Secure Comput.* (2015).

[100] HU, H., ET AL. Automatic Generation of Data-Oriented Exploits. In *USENIX Security* (2015).

[101] HU, H., ET AL. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE S&P* (2016).

[102] HU, H., ET AL. Enforcing Unique Code Target Property for Control-Flow Integrity. In *ACM CCS* (2018).

[103] HUANG, K., ET AL. The taming of the stack: Isolating stack data from memory errors. In *NDSS* (2022).

[104] HUANG, K., ET AL. Top of the Heap: Efficient Memory Error Protection of Safe Heap Objects. In *ACM CCS* (2024).

[105] ISMAIL, M., ET AL. VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks. In *ACM CCS* (2021).

[106] ISMAIL, M., ET AL. Tightly Seal Your Sensitive Pointers with PACTight. In *USENIX Security* (2022).

[107] ISPOGLOU, K. K., ET AL. Block Oriented Programming: Automating Data-Only Attacks. In *CCS* (2018).

[108] JACOBS, A., RAMPONI, C., ROELS, J., CRISPO, B., VLASCEANU, S., AMMAR, M., AND VOLCKAERT, S. N00B: Bounds checking for the masses. In *ACM CCS* (2026). To appear.

[109] JACOBS, A., AND VOLCKAERT, S. Not Quite Write: On the Effectiveness of Store-Only Bounds Checking. In *USENIX WOOT* (2024).

[110] JEON, Y., ET AL. Hextype: Efficient Detection of Type Confusion Errors for C++. In *ACM CCS* (2017).

[111] JIM, T., ET AL. Cyclone: a safe dialect of C. In *USENIX ATC* (2002).

[112] JIN, X., ET AL. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. In *IEEE S&P* (2022).

[113] JOHANNESMEYER, B., ET AL. Practical Data-Only attack generation. In *USENIX Security* (2024).

[114] JOLY, N., ET AL. Security Analysis of CHERI ISA. Tech. rep., Microsoft Security Response Center, October 2020. Accessed: 2025-06-05.

[115] JONES, R. W., AND KELLY, P. H. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *AADEBUG* (1997), pp. 13–26.

[116] KAYONDO, M., ET AL. MetaSafe: Compiling for Protecting Smart Pointer Metadata to Ensure Safe Rust Integrity. In *Proc. USENIX Security* (2024).

[117] KC, G. S., ET AL. Countering code-injection attacks with instruction-set randomization. In *CCS* (2003).

[118] KEMERLIS, V. P., ET AL. libdft: practical dynamic data flow tracking for commodity systems. In *ACM VEE* (2012).

[119] KHANDAKER, M., ET AL. Adaptive call-site sensitive control flow integrity. In *EuroS&P* (2019).

[120] KIL, C., ET AL. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *ACSAC* (2006).

[121] KIM, J., ET AL. TikTag: Breaking ARM's Memory Tagging Extension with Speculative Execution. *arXiv:2406.08719* (2024).

[122] KIRTH, P., ET AL. PKRU-safe: Automatically Locking Down the Heap Between Safe and Unsafe Languages. In *Proc. EuroSys* (2022).

[123] KOLOSICK, M., ET AL. Isolation Without Taxation: Near-Zero-Cost Transitions for WebAssembly and SFI. *Proc. ACM Program. Lang.* (2022).

[124] KONING, K., ET AL. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *DSN* (2016), IEEE.

[125] KROES, T., ET AL. Delta Pointers: Buffer Overflow Checks without the Checks. In *EuroSys* (2018).

[126] KUZNETSOV, V., ET AL. Code-pointer Integrity. In *OSDI* (2014).

[127] KUZNETZOV, V., ET AL. Code-pointer Integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. 2018.

[128] LARSEN, P., ET AL. SoK: Automated Software Diversity. In *IEEE S&P* (2014).

[129] LARSEN, P., AND SADEGHI, A.-R. *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. Association for Computing Machinery and Morgan & Claypool, 2018.

[130] LEE, B., ET AL. Preventing Use-after-free with Dangling Pointers Nullification. In *NDSS* (2015).

[131] LEE, B., ET AL. Type Casting Verification: Stopping an Emerging Attack Vector. In *USENIX Security* (2015).

[132] LEFEUVRE, H., ET AL. SoK: Software Compartmentalization. *arXiv* (2024).

[133] LEMAY, M., ET AL. Cryptographic Capability Computing. In *MICRO* (2021).

[134] LI, W., ET AL. PolyCruise: A Cross-Language Dynamic Information Flow Analysis. In *USENIX Security* (2022).

[135] LI, Y., ET AL. Finding cracks in shields: On the security of control flow integrity mechanisms. In *CCS* (2020).

[136] LI, Y., ET AL. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *ACM CCS* (2022).

[137] LILJESTRAND, H., ET AL. PAC It Up: Towards Pointer Integrity Using ARM Pointer Authentication. In *USENIX Security* (2019).

[138] LILJESTRAND, H., ET AL. Protecting the Stack with PACed Canaries. In *SysTEX* (2019).

[139] LILJESTRAND, H., ET AL. Color My World: Deterministic Tagging for Memory Safety. *arXiv preprint* (2022).

[140] LILJESTRAND, H., ET AL. A Viewpoint: Harmonizing the Diverse Memory Safety Fronts. *IEEE S&P* (2024).

[141] LIN, Z., ET AL. Dirtycred: Escalating privilege in linux kernel. In *ACM CCS* (2022).

[142] LIN, Z., ET AL. CAMP: Compiler and Allocator-based Heap Memory Protection. In *USENIX Security* (2024).

[143] LLORENTE-VAZQUEZ, O., ET AL. The Neverending Story: Memory Corruption 30 Years Later. In *CISIS/ICEUTE* (2022).

[144] LLVM COMMUNITY. Safe Stack. https://clang.llvm.org/docs/SafeStack.html, 2014. [Online; accessed 08-January-2025].

[145] LU, K., ET AL. Stopping Memory Disclosures via Diversification and Replicated Execution. *IEEE Trans. Dependable Secure Comput.* (2021).

[146] LU, T., AND WANG, J. Data-flow bending: On the effectiveness of data-flow integrity. *Computers & Security* (2019).

[147] MAAR, L., ET AL. Beyond the Edges of Kernel Control-Flow Hijacking Protection with HEK-CFI. In *ASIA CCS* (2024).

[148] MAAR, L., ET AL. SLUBStick: Arbitrary Memory Writes Through Practical Software Cross-Cache Attacks within the Linux Kernel. In *USENIX Security* (2024).

[149] MASHTIZADEH, A. J., ET AL. CCFI: Cryptographically enforced control flow integrity. In *CCS* (2015).

[150] MATSAKIS, N. D., ET AL. The Rust language. In *Proc. 2014 ACM SIGAda Ann. Conf. High Integrity Lang. Tech. (HILT)* (2014).

[151] MATSAKIS, N. D., AND KLOCK, F. S. The Rust Language. *ACM SIGAda Ada Letters* (2014).

[152] MERGENDAHL, S., ET AL. Cross-Language Attacks. In *NDSS* (2022).

[153] MICROSOFT. Control Flow Guard for platform security. https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, 2022. [Online; accessed 13-February-2023].

[154] MICROSOFT. Data Execution Prevention, 2022. [Online; accessed 13-February-2023].

[155] MIELE, A. Buffer Overflow Vulnerabilities in CUDA: A Preliminary Analysis. *Journal of Computer Virology and Hacking Techniques 12* (2016), 113–120.

[156] MIHRETIE, Y. E. Automatic Exploit Generation for Cross-Language Attacks. Master's thesis, MIT, 2022.

[157] MILBURN, A., ET AL. Safelnit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. In *NDSS* (2017).

[158] MILBURN, A., VAN DER KOUWE, E., AND GIUFFRIDA, C. Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation. In *IEEE S&P* (2022).

[159] NAGARAKATTE, S., ET AL. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI* (2009).

[160] NAGARAKATTE, S., ET AL. CETS: compiler enforced temporal safety for C. In *ISMM* (2010).

[161] NAM, M. J. FRAMER/Miu: Tagged Pointer-based Capability and Fundamental Cost of Memory Safety & Coherence (Position Paper). *arXiv preprint arXiv:2408.15219* (2024).

[162] NAM, M. J., ET AL. FRAMER: A Tagged-Pointer Capability System with Memory Safety Applications. In *ACSAC* (2019).

[163] NECULA, G. C., ET AL. CCured: Type-Safe Retrofitting of Legacy Software. *ACM TOPLAS* (2005).

[164] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices* (2007).

[165] NIU, B., AND TAN, G. Modular control-flow integrity. In *PLDI* (2014).

[166] NIU, B., AND TAN, G. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *CCS* (2014).

[167] NOVARK, G., AND BERGER, E. D. Dieharder: securing the heap. In *Proc. 17th ACM Conf. Comput. Commun. Secur. (CCS)* (2010).

[168] NUNES, I. D. O., ET AL. VRASED: A verified Hardware/Software Co-Design for Remote Attestation. In *USENIX Security* (2019).

[169] NUNES, I. D. O., ET AL. Dialed: Data Integrity Attestation for Low-End Embedded Devices. In *DAC* (2021).

[170] NUNES, I. D. O., ET AL. Tiny-CFA: A Minimalistic Approach for Control-Flow Attestation using Verified Proofs of Execution. *DATE* (2021).

[171] NYMAN, T. *Toward hardware-assisted run-time protection*. PhD thesis, Aalto University, 2020.

[172] NYMAN, T., ET AL. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *RAID* (2017).

[173] ONE, A. Smashing the Stack for Fun and Profit. *Phrack Magazine* (1996).

[174] ORTHEN, B., ET AL. SoftBound+CETS Revisited: More Than a Decade Later. In *EuroSec* (2024).

[175] ÖSTERLUND, S., ET AL. kMVX: Detecting kernel information leaks with multi-variant execution. In *ASPLOS* (2019).

[176] PAPAEVRIPIDES, M., AND ATHANASOPOULOS, E. Exploiting mixed binaries. *ACM Trans. Priv. Secur.* (2021).

[177] PAPPAS, V., ET AL. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *IEEE S&P* (2012).

[178] PARK, S.-O., ET AL. Mind control attack: Undermining deep learning with GPU memory exploitation. *Computers & Security* (2021).

[179] PAX TEAM. Address Space Layout Randomization (ASLR), 2003. [Accessed: 2025-05-31].

[180] PAX TEAM. Non-Executable Pages Design & Implementation, 2003. [Online; accessed 18-February-2023].

[181] PAYER, M. Comparing Control-Flow Integrity in CFI, CPS, and CPI. https://nebelwelt.net/blog/2014/1007-CFICPSCPIdiffs.html, 2014.

[182] PHRACK MAGAZINE. Bypassing PaX ASLR Protection. *Phrack* (2002).

[183] QIN, B., ET AL. Understanding memory and thread safety practices and issues in real-world Rust programs. In *PLDI* (2020).

[184] RAJASEKARAN, P., ET AL. CoDaRR: Continuous Data Space Randomization against Data-only Attacks. In *ACM ASIA CCS* (2020).

[185] RAVICHANDRAN, J., ET AL. PACMAN: attacking ARM pointer authentication with speculative execution. In *ISCA* (2022).

[186] ROELS, J., ET AL. CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World. In *EuroSec* (2025).

[187] ROEMER, R., ET AL. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security* (2012).

[188] ROGLIA, G. F., ET AL. Surgically returning to randomized lib (c). In *ACSAC* (2009).

[189] RUDD, R., ET AL. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *NDSS* (2017).

[190] RUWASE, O., AND LAM, M. S. A Practical Dynamic Buffer Overflow Detector. In *NDSS* (2004).

[191] SALAMAT, B., ET AL. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *EuroSys* (2009).

[192] SCHELFHOUT, A., ET AL. Diagnosing and neutralizing address-sensitive behavior in multi-variant execution systems. In *EuroSec* (2025).

[193] SCHUSTER, F., ET AL. Evaluating the effectiveness of current anti-ROP defenses. In *RAID* (2014).

[194] SCHUSTER, F., ET AL. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE S&P* (2015).

[195] SELIFONOV, D. Drinking from LETHE: New methods of exploiting and mitigating memory corruption vulnerabilities. Presented at DEF CON, 2015. https://media.defcon.org/DEF%20CON%2023/DEF%20CON%2023%20presentations/DEF%20CON%2023%20-%20Daniel-Selifonov-Drinking-from-LETHE.pdf.

[196] SEO, J., ET AL. Sfitag: Efficient Software Fault Isolation with Memory Tagging for ARM Kernel Extensions. In *Asia CCS* (2023).

[197] SEO, J., ET AL. ZOMETAG: Zone-based Memory Tagging for Fast, Deterministic Detection of Spatial Memory Violations on ARM. *IEEE TIFS* (2023).

[198] SEREBRYANY, K., ET AL. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Security* (2012).

[199] SEREBRYANY, K., ET AL. Memory tagging and how it improves C/C++ memory safety. *arXiv preprint arXiv:1802.09517* (2018).

[200] SERNA, F. J. The Info Leak Era on Software Exploitation. *Black Hat USA* (2012).

[201] SHACHAM, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proc. ACM CCS* (2007).

[202] SHACHAM, H., ET AL. On the Effectiveness of Address-Space Randomization. In *CCS* (2004).

[203] SHARMA, A. This New Android 14 Feature May Be Meant for the Pixel 8. https://www.androidauthority.com/android-14-advanced-memory-protection-3281197/, 2023. [Online; accessed 18-February-2023].

[204] SNOW, K. Z., ET AL. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE S&P* (2013).

[205] SONG, C., ET AL. Enforcing Kernel Security Invariants with Data Flow Integrity. In *NDSS* (2016).

[206] SONG, C., ET AL. HDFI: Hardware-assisted data-flow isolation. In *SP* (2016).

[207] SONG, D., ET AL. SoK: Sanitizing for Security. In *IEEE S&P* (2019).

[208] SPAFFORD, E. H. The Internet Worm Program: An Analysis. *SIGCOMM Comput. Commun. Rev.* (1989).

[209] STRACKX, R., ET AL. Breaking the memory secrecy assumption. In *Proc. 2nd Eur. Workshop Syst. Security (EUROSEC)* (2009).

[210] SUN, Z., ET AL. OAT: Attesting Operation Integrity of Embedded Devices. In *Proc. IEEE S&P* (2020).

[211] SZEKERES, L., ET AL. SoK: Eternal War in Memory. In *IEEE S&P* (2013).

[212] TANG, A., ET AL. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM CCS* (2015).

[213] TICE, C., ET AL. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security* (2014).

[214] TICE, C., ET AL. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security* (2014).

[215] TOM GARRISON. Intel CET Answers Call to Protect Against Common Malware Threats. https://newsroom.intel.de/editorials/intel-cet-answers-call-to-protect-against-common-malware-threats/, 2020. [Online; accessed 13-February-2023].

[216] ULLAH, S., AND RASHID, A. Security Implications of the Morello Platform: An Empirical Threat Model-Based Analysis. *ACM Trans. Priv. Sec.* (2025).

[217] UNTERGUGGENBERGER, M., ET AL. Cryptographically Enforced Memory Safety. In *CCS* (2023).

[218] VAHLDIEK-OBERWAGNER, A., ET AL. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security* (2019).

[219] VAN DER KOUWE, E., ET AL. Dangsan: Scalable Use-After-Free Detection. In *EuroSys* (2017).

[220] VAN DER KOUWE, E., ET AL. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In *Proc. 34th ACM Comput. Syst. Appl. Conf. (ACSAC)* (2018).

[221] VAN DER VEEN, V., ET AL. Memory Errors: The Past, the Present, and the Future. In *RAID* (2012).

[222] VOLCKAERT, S. *Advanced Techniques for multi-variant execution*. PhD thesis, Ghent University, 2015.

[223] VOLCKAERT, S., ET AL. Cloning Your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution. *IEEE Trans. Dependable Secure Comput.* (2016).

[224] VOLCKAERT, S., ET AL. Secure and efficient application monitoring and replication. In *USENIX ATC* (2016).

[225] VOULIMENEAS, A., ET AL. You shall not (by) pass! practical, secure, and fast pku-based sandboxing. In *EuroSys* (2022).

[226] WAHBE, R., ET AL. Efficient software-based fault isolation. In *OSDI* (1993).

[227] WANG, R., ET AL. S2malloc: Statistically Secure Allocator for Use-After-Free Protection And More. In *DIMVA* (2024).

[228] WANG, Y., ET AL. Opportunistic data flow integrity for real-time cyber-physical systems using worst case execution time reservation. In *USENIX Security* (2024).

[229] WARTELL, R., ET AL. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy x86 Binary Code. In *ACM CCS* (2012).

[230] WATSON, R. N., ET AL. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *SP* (2015).

[231] WICKMAN, B., ET AL. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *USENIX Security* (2021).

[232] WIEBING, S., ET AL. InSpectre Gadget: Inspecting the residual attack surface of cross-privilege Spectre v2. In *USENIX Security* (2024).

[233] WOODRUFF, J., ET AL. The CHERI capability model: Revisiting RISC in an age of risk. *ACM SIGARCH Comput. Archit. News* (2014).

[234] XIE, D., ET AL. BridgeRouter: Automated Capability Upgrading of Out-Of-Bounds Write Vulnerabilities to Arbitrary Memory Write Primitives in the Linux Kernel. In *IEEE S&P* (2025).

[235] XU, H., ET AL. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Transactions on Software Engineering and Methodology* (2021).

[236] XU, S., ET AL. In-fat Pointer: Hardware-assisted Tagged-pointer Spatial Memory Safety Defense with Subobject Granularity Protection. In *ASPLOS* (2021).

[237] YAGEMANN, C., CHUNG, S. P., SALTAFORMAGGIO, B., AND LEE, W. {PUMM}: Preventing {Use-After-Free} using execution unit partitioning. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023), pp. 823–840.

[238] YANG, F., ET AL. Endokernel: A Thread Safe Monitor for Lightweight Subprocess Isolation. In *Proc. USENIX Security* (2024).

[239] YE, H., ET AL. VIPER: Spotting Syscall-Guard variables for Data-Only attacks. In *USENIX Security* (2023).

[240] YE, H., ET AL. Too Subtle to Notice: Investigating Executable Stack Issues in Linux Systems. In *NDSS* (2025).

[241] YOUNAN, Y. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS* (2015).

[242] YOUNAN, Y., ET AL. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *ASIACCS* (2010).

[243] YU, Z., ET AL. ShadowBound: Efficient Heap Memory Protection Through Advanced Metadata Management and Customized Compiler Optimization. In *USENIX Security* (2024).

[244] ZHANG, M., ET AL. eXecutable-Only-Memory-Switch (XOM-Switch): Hiding Your Code From Advanced Code Reuse Attacks in One Shot. Presented at Black Hat Asia, 2018. https://i.blackhat.com/briefings/asia/2018/asia-18-Zhang-Liu-Xom-switch--v1.3.pdf.

[245] ZHANG, M., AND SEKAR, R. Control Flow Integrity for COTS Binaries. In *USENIX Security* (2013).

A Further Mapping of Defenses

Table 6 provides an extended mapping of additional memory corruption defenses to the exploit graph model (Figure 2), following the same methodology as in Table 4.

B Defense Mapping Examples

Our paper primarily focuses on the motivation, construction methodology, and utility of our graph-based model. This appendix instead shows hands-on examples about how to use our model using the Python interface.

Listing 2 shows exactly how the EffectiveSan-bounds [70] defense is mapped. Defenses are implemented as `static` or `class` methods that declaratively specify which edges they inhibit. They do so by looking up the relevant edge using shorthand names for any source and sink nodes (`'v_dp_reg'` and `'dp_to_oob'` in this case), and marking them as inhibited. Any fragility query can now use the resulting Python method to apply the protection of EffeciveSan-bounds to its modeled set of defenses, facilitating composability for multi-defense analysis.

Also note that each defense optionally comes with a set of informative annotations, like a BibTex citation and display name. We use these to automatically generate informative summaries like Table 4, which ensures that the paper content precisely matches the actual implementation and modeled defense.

The method additionally contains a `@threatmodel` annotation, which is simply a list of other Python methods (often other defenses). These can be used to separately indicate the exact threat model under which the defense assumes it is operating. We take these into account during our automated design-level fragility testing Section 5, to ensure that we only find bypasses that fit the threat model of the authors. Note that, in this process, we encountered many papers without clear threat models, or that omit "obvious" assumptions like the presence of non-writable code pages. We tried to be as pragmatic as possible in this case, to maximize the relevance of our results. Nevertheless, our model helps to clarify such cases by providing a standardized notation and platform that can avoid such ambiguity in the future.

Defense mappings are not only declarative, they are also composable. Listing 3 illustrates this through the full EffectiveSan model, which essentially consists of the `-type` subset, the `-bounds` subset (Listing 2), with the temporal safety edges

Table 6: Extended mapping of additional memory corruption defenses to attack steps in the exploit graph.

Defense	Strategy	Category	Inhibited Edges or Knowledge	Assumed Threat Model	Attacker Goals			Execution Control		
					Info. Disclosure.	Access Control	Arbitrary Computation	Func.-Gran. CF Hijack	Instr.-Gran. CF Hijack	ACE
Baggy Bounds [5]	MS	Spatial	3	1 4-5						
Baggy Bounds (64-bit) [5]	MS	Spatial	3	1 4-5						
PTAuth [78]	MS	Temporal	12-13	1 3-4						
PUMM [237]	MS	Temporal	5	1 3-4						
CAMP [142]	MS	Spatial, Temporal	3 5	1 4						
ShadowBound [243]	MS	Spatial	3	1 4-5						
ShadowBound-MarkUs [243]	MS	Spatial, Temporal	3 5	1 4						
ShadowBound-FFMalloc [243]	MS	Spatial, Temporal	1-3	1 4						
ShadowBound-PUMM [243]	MS	Spatial, Temporal	3 5	1 4						
DANGNULL [130]	MS	Temporal	5	1 3-4						
Oscar [59]	MS	Temporal	12-13	1 3-4						
CAVER [131]	MS	Miscasts	4	1 3 5						
Type++ [17]	MS	Miscasts	4	1 3 5						
TASR [25]	IoD	Code/Data Randomization	6 CLK	4 3-4	✓	✓	✓	✓		
VIP-CFI [105]	IoU	CFI		4-5 12 3-4	✓	✓	✓	✓	✓	✓
VIP-CPI [105]	IoD, IoU	CPI	5	4-5 12 3-4	✓	✓	✓	✓	✓	✓
ARM BTI [12]	IoU	CFI	2 3	4-5 12 3-4	✓	✓	✓	✓		
PAC-RET [11]	IoU	CFI	5 12 1 4-5 4	7 10 4 6 11 3-4	✓	✓	✓			
Clang Standard Protection (ARM) [11, 12]	IoU	CFI	1-2 4-5 3-4 5 12	4 3-4	✓	✓	✓	✓		
CFA+ [8]	IoU, PED	CFI	1-6 3-4	4 3-4	✓	✓	✓			

Legend: **CF:** Control Flow, **ACE:** Arbitrary Code Execution, **MS:** Memory Safety, **IoD:** Isolation of Data, **IoU:** Integrity of Use, **PED:** Post-Exploit Detection, **CLK:** Code Layout Knowledge, **CCK:** Code Content Knowledge, **DLK:** Data Layout Knowledge, **MVX:** Multi-Variant eXecution, **CPI:** Code Pointer Integrity, **CFI:** Control Flow Integrity, **DFI:** Data Flow Integrity, ✓ Attacker has achieved this goal.

```
@staticmethod
@citation("effectivesan")
@threatmodel([no_uninit, no_miscasts, no_temporal
    ])
@prettyname("EffectiveSan-bounds")
def EffectiveSan_bounds(graph: ExploitGraph,
    explorer: Explorer):
   explorer.inhibit(
      graph.find_edge('v_dp_reg', 'dp_to_oob') #
          essentially lowfat for bounds
   )
```

Listing 2: Python model mapping for EffectiveSan-bounds [70].

on top. Composability is additionally supported by specifying other defenses in the threat model, as shown for in Mon CHÉRI [94] Listing 4, which assumes the presence of the CHERI capability system as a platform for its uninitialized read detection.

```
@classmethod
@threatmodel([no_uninit])
@citation("effectivesan")
def EffectiveSan(self, graph: ExploitGraph, explorer
    : Explorer):
   self.EffectiveSan_type(graph, explorer)
   self.EffectiveSan_bounds(graph, explorer)
   explorer.inhibit(
      graph.find_edge('dp_to_dangling', '
          memacc_read'),
      graph.find_edge('dp_to_dangling', '
          memacc_write'),
   )
```

Listing 3: Python model mapping for complete EffectiveSan [70].

```
@staticmethod
@threatmodel([CHERI,Cornucopia,no_miscasts])
@citation("gulmez2025moncheri")
@prettyname(R"Mon CH{\'E}RI")
def MonCheri(graph: ExploitGraph, explorer: Explorer
    ):
   explorer.inhibit(graph.find_edge('dp_to_uninit',
       'memacc_read'))
```

Listing 4: Python model mapping for Mon CHÉRI [94].