NDSS 2026

Breaking Isolation

A New Perspective on Hypervisor Exploitation via Cross-Domain Attacks

Authors

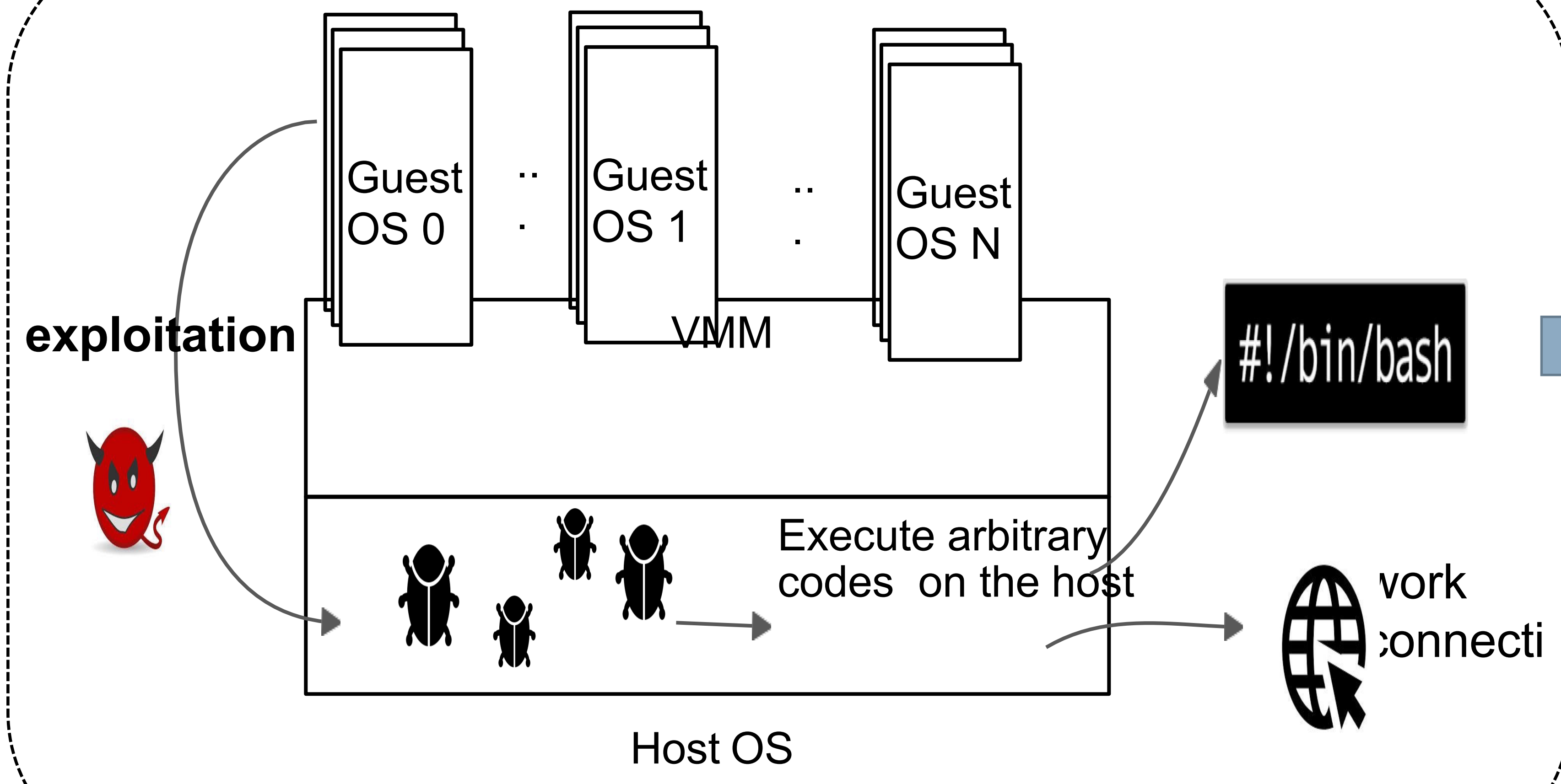
Gaoning Pan, Yiming Tao, Qinying Wang,
Chunming Wu, Mingde Hu, Yizhi Ren, Shouling Ji

Affiliations

Hangzhou Dianzi University
Zhejiang University
EPFL
Zhejiang Provincial Key Laboratory of Sensitive Data Security and Confidentiality Governance

Hypervisor Security Landscape

Virtual Machine Escape



Public cloud compromise

Hypervisor Security Landscape

QEMU CVE Classification (2019-2024)

Vulnerability Category	Ptr. Corr.	Data-only Corr.	No Corr.	Total
Use-After-Free	12	1	2	15
OOB Write	18	12	0	30
OOB Read	0	0	14	14
Integer Overflow	1	5	1	7
Uninitialized Variable	1	0	0	1
Information Leak	0	0	13	13
Logic/Crash and Others	0	0	54	54
Subtotal	**32**	**18**	**84**	**134**
Percentage	**23.9%**	**13.4%**	**62.7%**	**100.0%**

Ptr. Corr. = vulnerabilities that may alter pointer values. Data-only Corr. = vulnerabilities affecting data but not pointers. No Corr. = vulnerabilities with no corruptive effect on memory.

Finding: Pointer corruption is prevalent in hypervisor vulnerabilities

Why Exploiting Pointer Corruption is Difficult

```c
// guest-controlled offset
void usbredir_buffered_bulk_packet(...,
    ↪ uint8_t *data, size_t data_len, ...) {
    size_t i = choose_offset(...);
    ...
    bufp_alloc(dev, data + i, len, status, ep,
        ↪  data);   // interior ptr
}

int bufp_alloc(USBRedirDevice *dev, uint8_t *
    ↪ data, uint16_t len, ...) {
    ...
    if (bufpq_should_drop(dev, ep)) {
        free(data); // free(data + i): not
            ↪ chunk base
        return -1;
    }
    ...
}
```

user controlled

arbitrary free

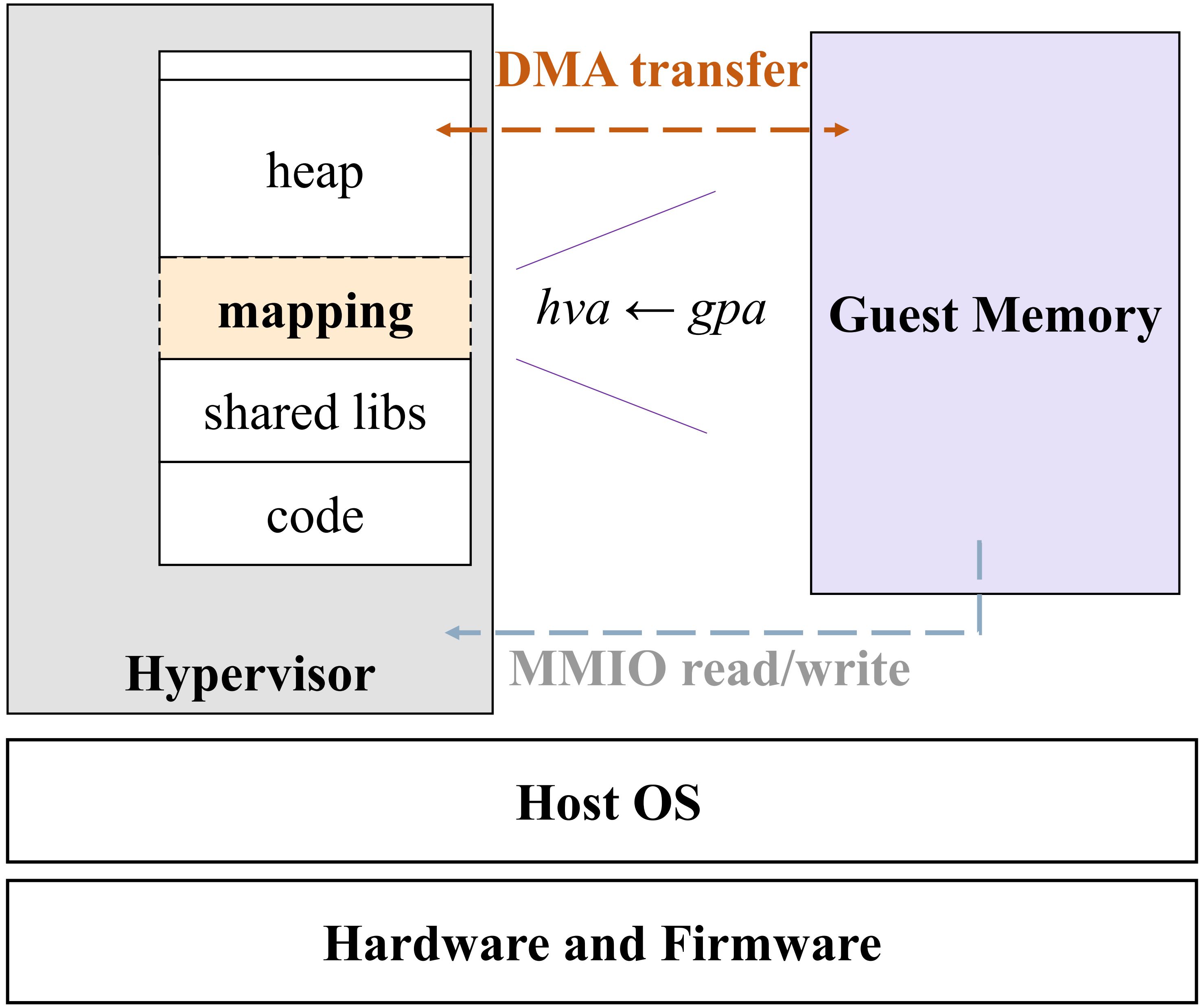
1. Scarcity of Exploitable Structures

(What objects can actually be freed?)

2. ASLR-Induced Address Uncertainty

(Where does it point to?)

Key Insight: Guest-Host **Weak Memory Isolation**



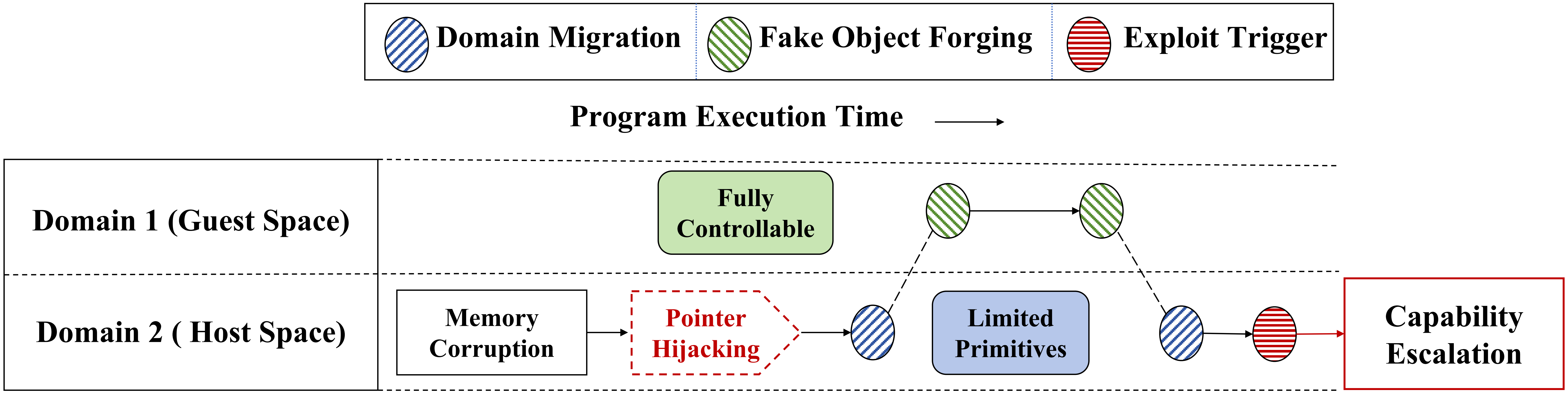
Overlooked Attack Surface

→ **Guest memory** is **fully attacker-controlled**

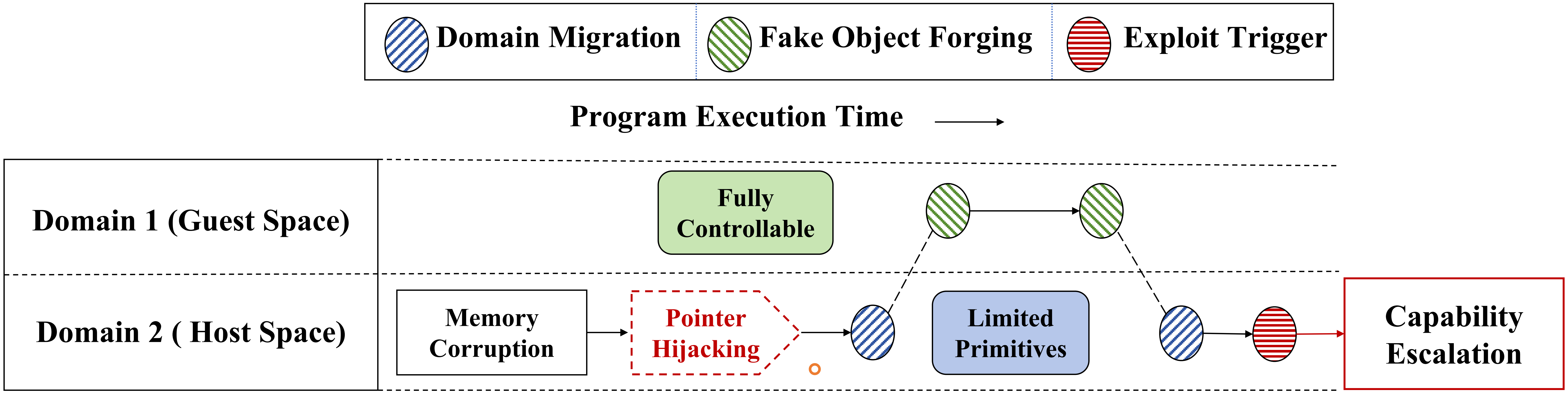
→ Host can **freely dereference** guest memory

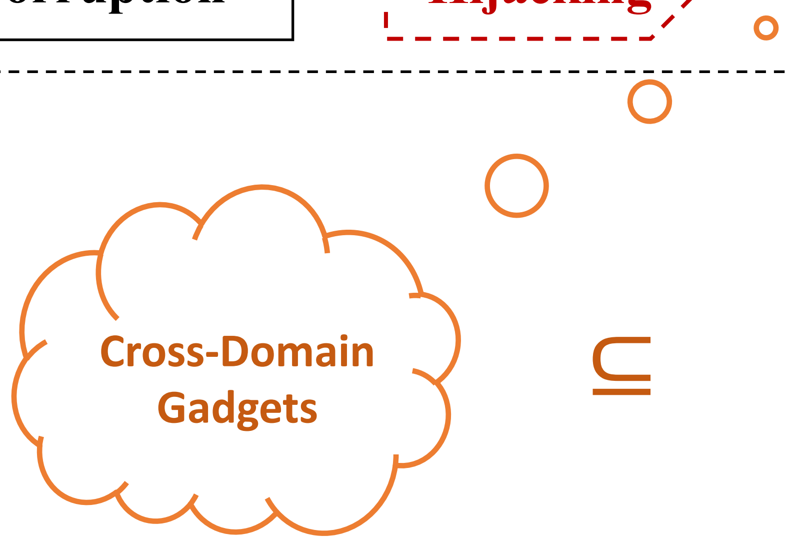
→ Creates **new primitive** for capability escalation

Our Exploitation: Cross Domain Attack



Our Exploitation: Cross Domain Attack





Category	Translation Functions

QEMU	address_space_map();
	address_space_unmap();
	address_space_read_full();
	address_space_read();
	address_space_write();

Virtualbox	PGMPhysWrite(); PGMPhysRead(); PGMR3PhysBulkGCPhys2CCPtrExternal(); PGMR3PhysBulkGCPhys2CCPtrReadOnlyExternal(); PGMR3PhysGCPhys2CCPtrExternal();

Code Sample of the CDA

```
// Guest (Attacker)

fake = guest_alloc_page();
    ↪ (1) allocate fake object
mmio_write(MMIO_ADDR, map_gpa(fake));
    ↪ (2) send fake object's GPA
...
fake->ops = final_attacker_ops;
    ↪ (5) modify fake object to
          finalize the exploit
```

Attacker's payload

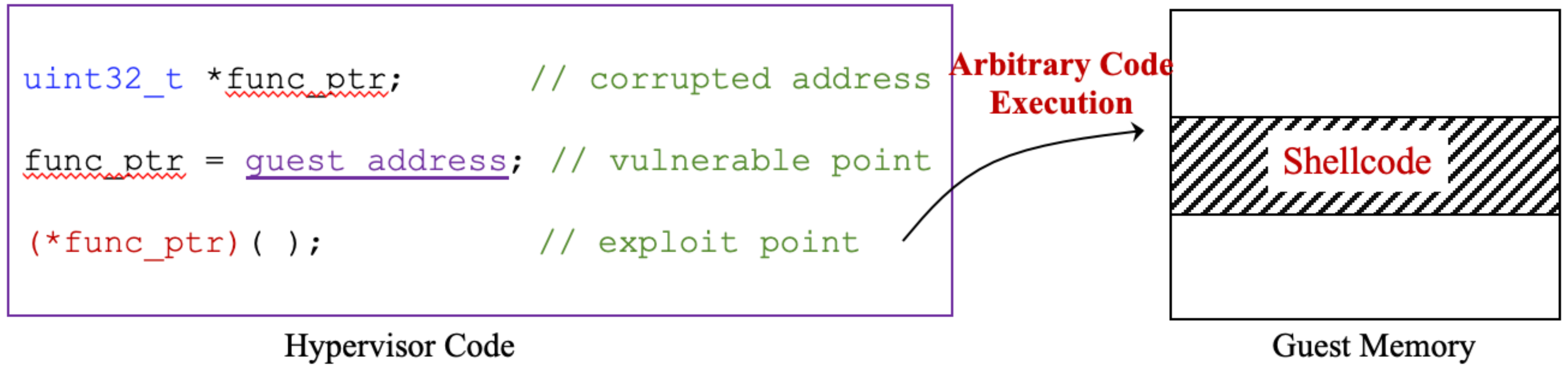
```
// Host (Hypervisor)

s->ptr = (Req *)gpa_to_hva(gpa);
    ↪ (3) vulnerability overwrites
          pointer to fake object
qemu_free(s->ptr);
    ↪ (4) host uses attacker object
```

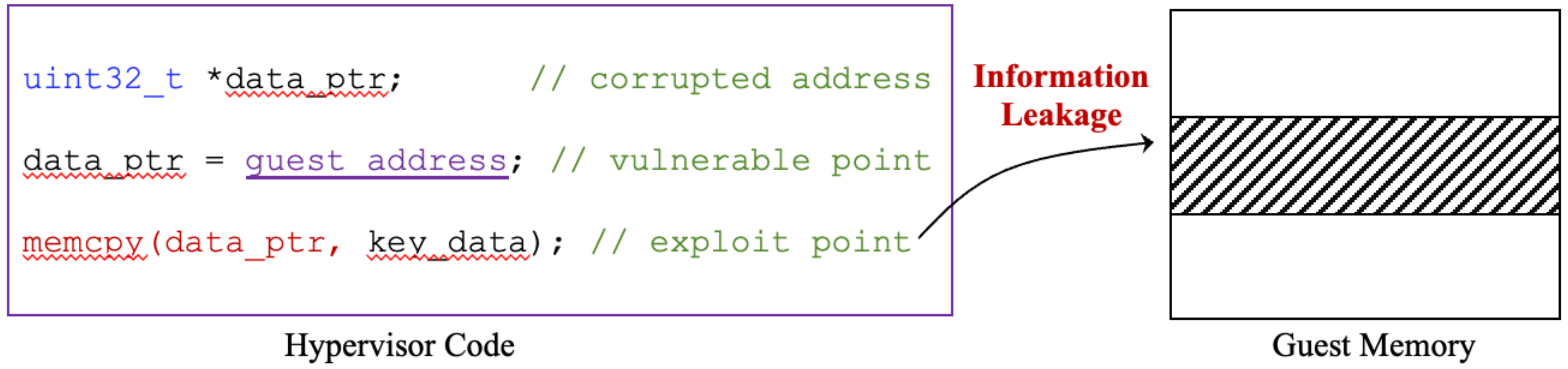
Vulnerable code

CDA Variants

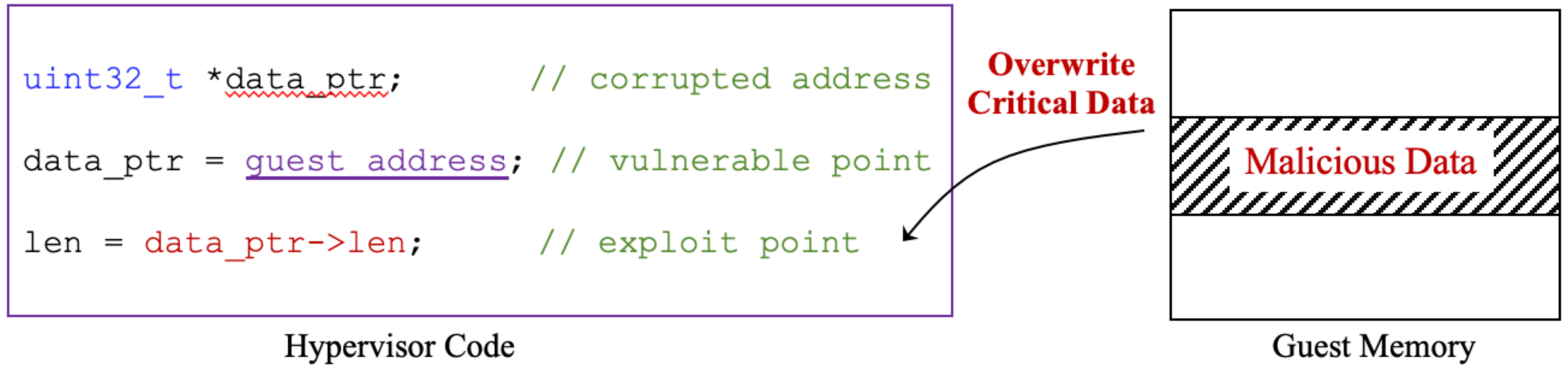
CDA[A]: Arbitrary Code Execution



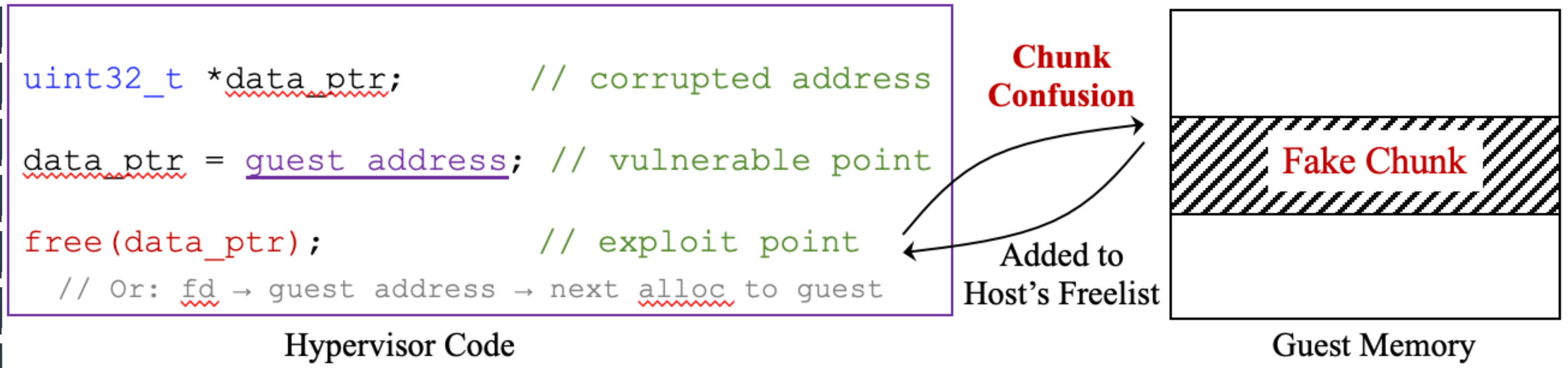
CDA[I]: Information Leakage



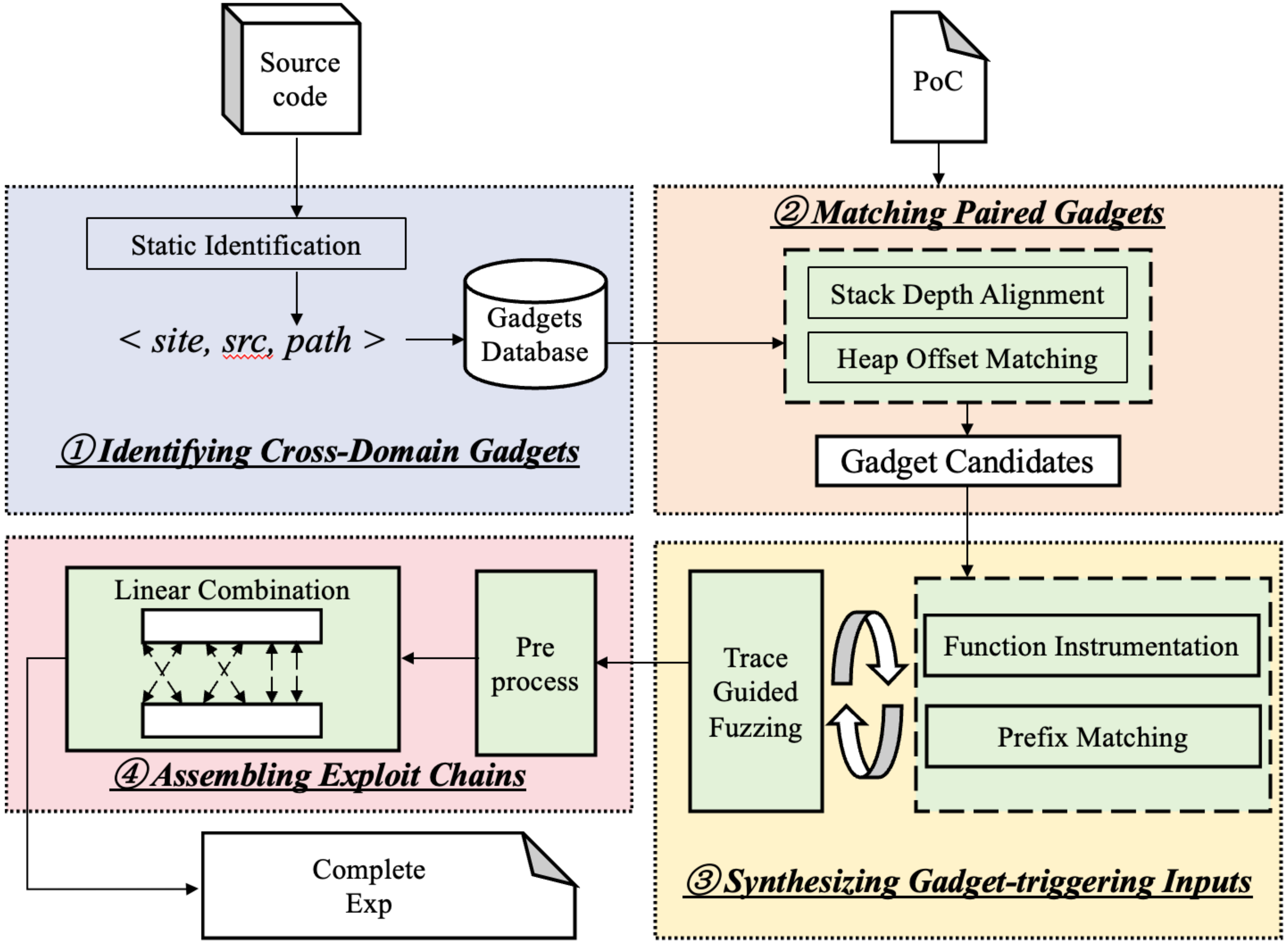
CDA[O]: Critical Data Overwriting



CDA[C]: Chunk Confusion



Automated CDA Framework



Input

- → Hypervisor source code
- → PoC triggering pointer corruption

Output

- → Redirecting corrupted pointer to attacker-controlled guest memory
- → Achieving one of four CDA variants

1. Identifying Cross-Domain Gadgets

Definition of CDA Gadgets

$$G = \{(site, src, path) \mid site \in S, src \in T, path \in P\}$$

site: Program location of GPA-to-HVA translation

path: Static call chain from guest interface to translation site

src: Guest-controllable operand flowing into translation

Static Analysis Pipeline

1. Locate Translation Sites
2. Trace GPA Origin
3. Extract Call Chains
4. Build Database

Gadget Family	Upper Function	Translation Function	HVA Variable	GPA Source Field	Trigger Type	Call Path
DMA gadget	dma_memory_write	address_space_write	ram_ptr	s→tx_descriptor	MMIO	nvme_mmio_write → nvme_process_db → stl_le_pci_dma → stl_le_dma → dma_memory_write

2. Matching Paired Gadgets

Algorithm 1: Paired Gadget Matching Strategy

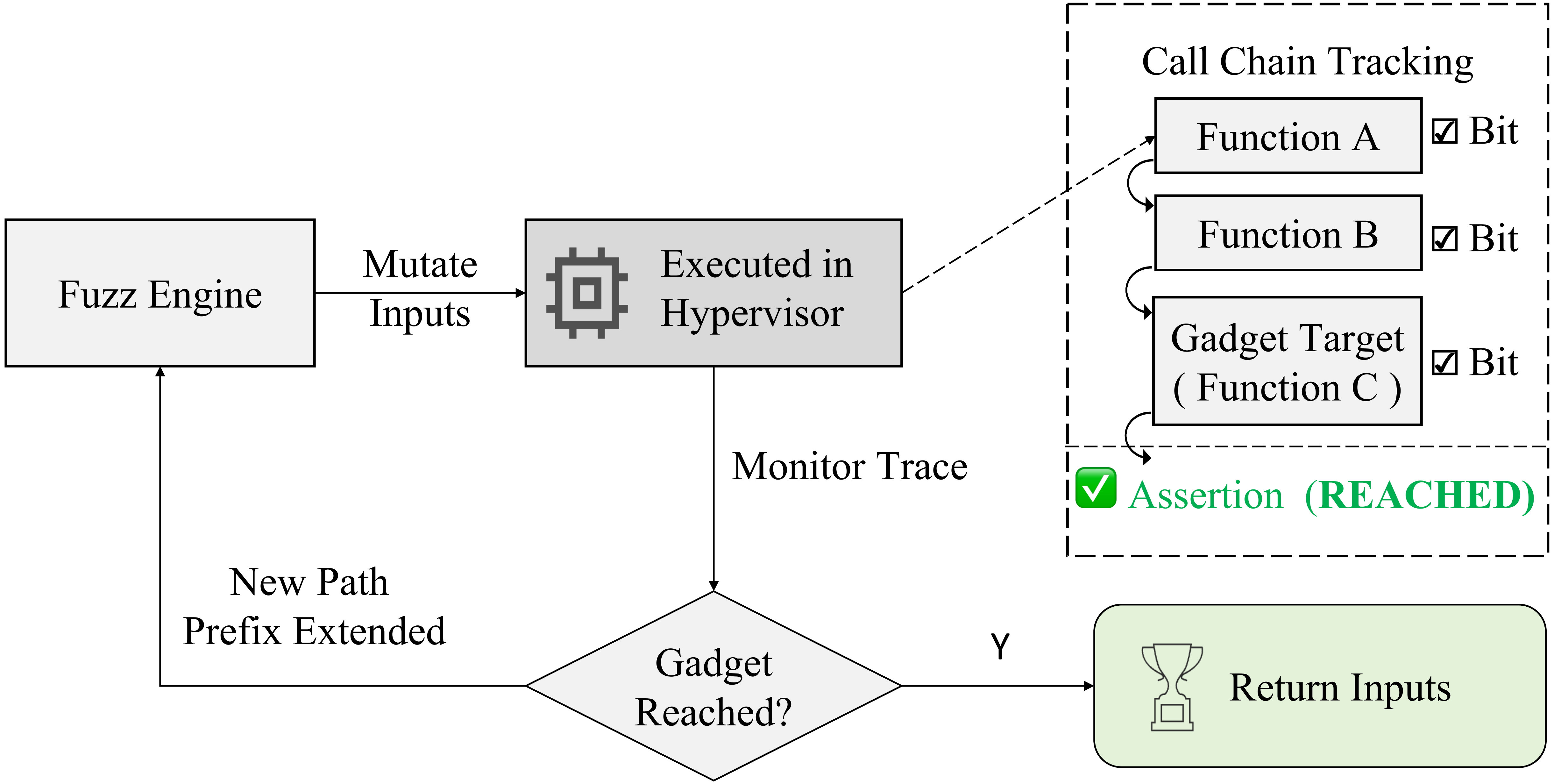
Input: Gadget database $\mathcal{G}$, corrupted pointer metadata M, pointer region $R \in \{\text{stack}, \text{heap}\}$

Output: Set of matched gadgets $\mathcal{G}_{\text{match}}$

$\mathcal{G}_{\text{match}} \leftarrow \emptyset$;
if $R = \text{stack}$ **then**
  foreach $g = (site, src, path) \in \mathcal{G}$ **do**
    $g.depth \leftarrow \text{Length}(path)$;
  $p_{\text{depth}} \leftarrow M.\text{stack\_depth}$;
  foreach $g \in \mathcal{G}$ **do**
    if $g.depth = p_{\text{depth}}$ **then**
      $\mathcal{G}_{\text{match}} \leftarrow \mathcal{G}_{\text{match}} \cup \{g\}$;
else if $R = \text{heap}$ **then**
  foreach $g \in \mathcal{G}$ **do**
    if $\text{IsStoredAsStructField}(g)$ **then**
      $(g.size, g.offset) \leftarrow \text{ExtractStructInfo}(g)$;
  $(s_{\text{size}}, o_{\text{offset}}) \leftarrow M.\text{heap\_layout}$;
  foreach $g \in \mathcal{G}$ **do**
    if $g.size = s_{\text{size}}$ **and** $g.offset = o_{\text{offset}}$ **then**
      $\mathcal{G}_{\text{match}} \leftarrow \mathcal{G}_{\text{match}} \cup \{g\}$;
return $\mathcal{G}_{\text{match}}$

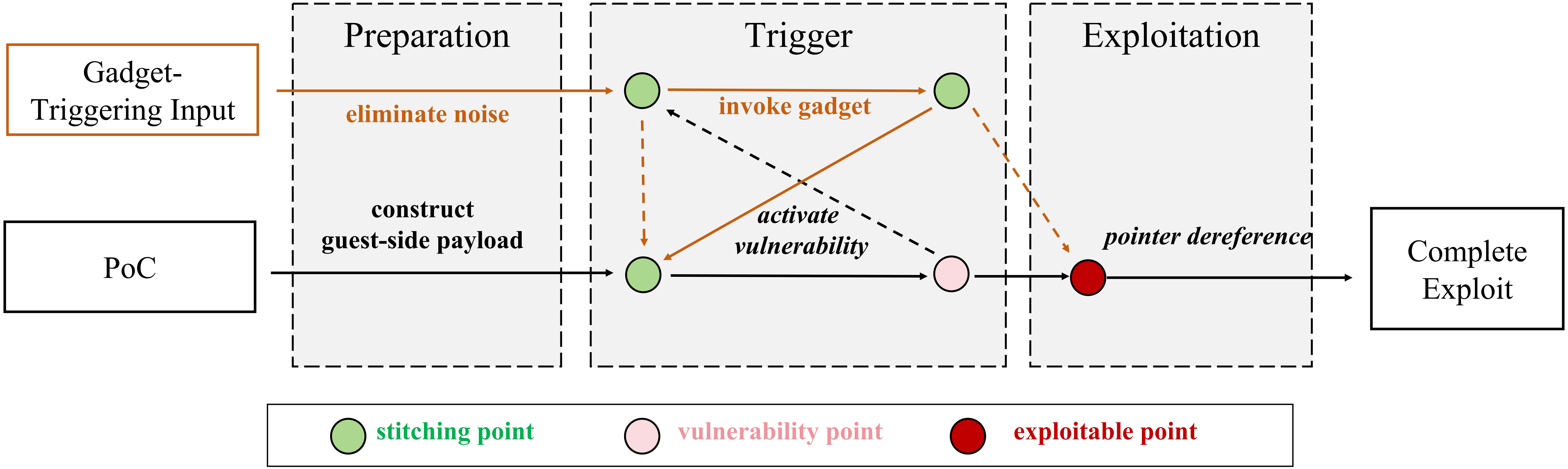
Insight: *pair a vulnerability with the gadget whose residual guest-HVA pointer is spatially aligned with the corrupted pointer*

3. Synthesizing Gadget-triggering Inputs



Trace-guided fuzzing workflow

4. Assembling Exploit Chains

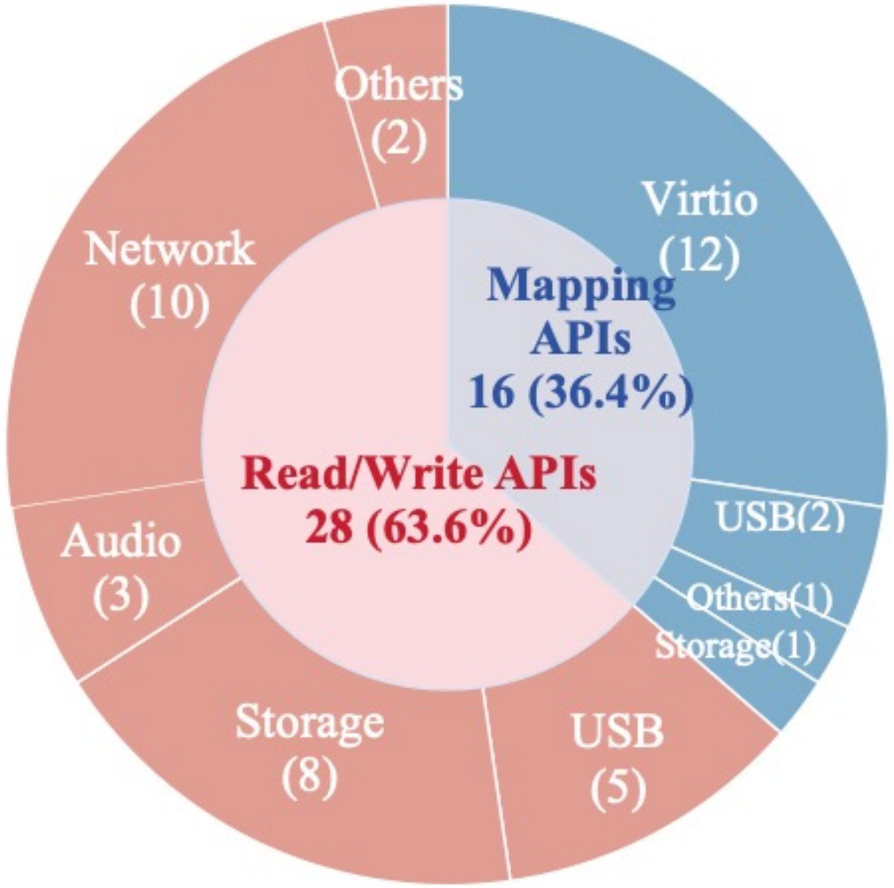


Evaluation

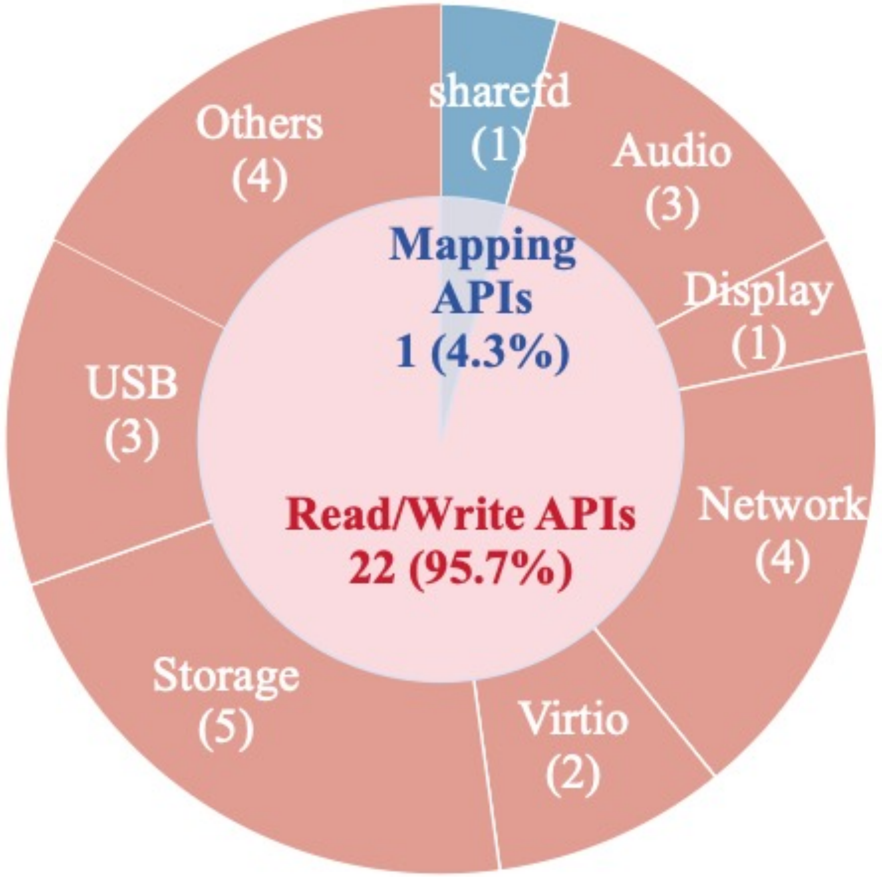
- **RQ1**: How prevalent are cross-domain gadgets, and what is their distribution within hypervisors?
- **RQ2**: How frequently do cross-domain gadgets produce guest–HVA pointers?
- **RQ3**: How does CDA perform in actual exploit scenarios?

Evaluation: Gadget Prevalence

Translation Function Distribution



QEMU



VirtualBox

Key Finding: GPA-to-HVA translation is **deeply embedded** across hypervisor device-emulation code, making CDA broadly applicable.

Evaluation: Gadget Prevalence

Gadget Family	Upper Function	Translation Function	HVA Variable	GPA Source Field	Trigger Type	Count
DMA gadget	dma_memory_map	address_space_map	ad→lst	AHCIPortRegs→fis_addr	MMIO (4), BH (0)	4
	pci_dma_map	address_space_map	ring→page	txd.addr	MMIO (10), BH (0)	10
	dma_memory_read	address_space_read_full	ram_ptr	s→tx_descriptor	MMIO (81), BH (21)	102
	dma_memory_write	address_space_write	ram_ptr	s→tx_descriptor	MMIO (91), BH (16)	107
USB gadget	usb_packet_map	address_space_map	packet→iov	sgl→sg[num_sg].base	MMIO (3), BH (11)	14
	get_dwords	address_space_read_full	ram_ptr	q→qhaddr	MMIO (1), BH (35)	36
	put_dwords	address_space_write	ram_ptr	q→qhaddr	MMIO (2), BH (44)	46
	xhci_dma_read_u32s	address_space_read_full	ram_ptr	sctx→pctx	MMIO (22), BH (8)	30
	xhci_dma_write_u32s	address_space_write	ram_ptr	sctx→pctx	MMIO (17), BH (4)	21
	xhci_write_event	address_space_write	ram_ptr	intr→er_start	MMIO (17), BH (5)	22
Virtio gadget	virtqueue_map_desc	address_space_map	iov[num_sg].iov_base	desc[num_sg].addr	MMIO (40), BH (16)	56
	virtio_gpu_create_mapping_iov	address_space_map	iov[num_sg].iov_base	desc[num_sg].addr	MMIO (0), BH (2)	2
Display gadget	cpu_physical_memory_map	address_space_map	data	s→dispc.l[0].addr[0]	MMIO (1), BH (0)	1
Block device gadget	dma_blk_cb	address_space_map	dbs→iov	req→sg.qsg	MMIO (4), BH (6)	10
SCSI gadget	lsi_mem_read	address_space_read	ram_ptr	s→dsp	MMIO (4), BH (0)	4
	lsi_mem_write	address_space_write	ram_ptr	s→dsp	MMIO (4), BH (0)	4
PCI gadget	pci_dma_read	address_space_read_full	ram_ptr	r→bdbar	MMIO (59), BH (120)	179
	pci_dma_write	address_space_write	ram_ptr	desc.buffer_addr	MMIO (42), BH (22)	64
SDHCI gadget	sdhci_do_adma	address_space_read_full	ram_ptr	dscr.addr	MMIO (12), BH (2)	14
	sdhci_sdma_transfer_multi_blocks	address_space_read_full	ram_ptr	s→sdmasysad	MMIO (9), BH (1)	10
	sdhci_sdma_transfer_multi_blocks	address_space_write	ram_ptr	desc.buffer_addr	MMIO (9), BH (1)	10
Total						**772**

Key Finding: 772 gadget instances in QEMU, clustered into 8 major gadget families.

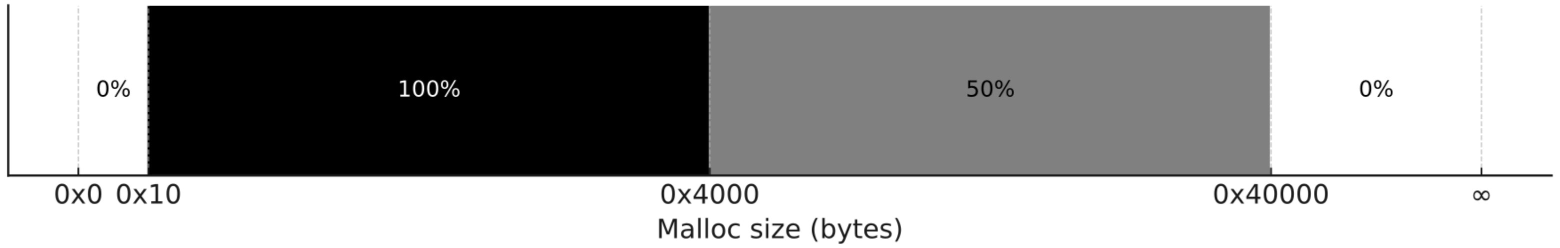
Evaluation: Pointer Presence

Stack Coverage

(a) MMIO entry

(b) Timer/BH entry.

Heap Coverage



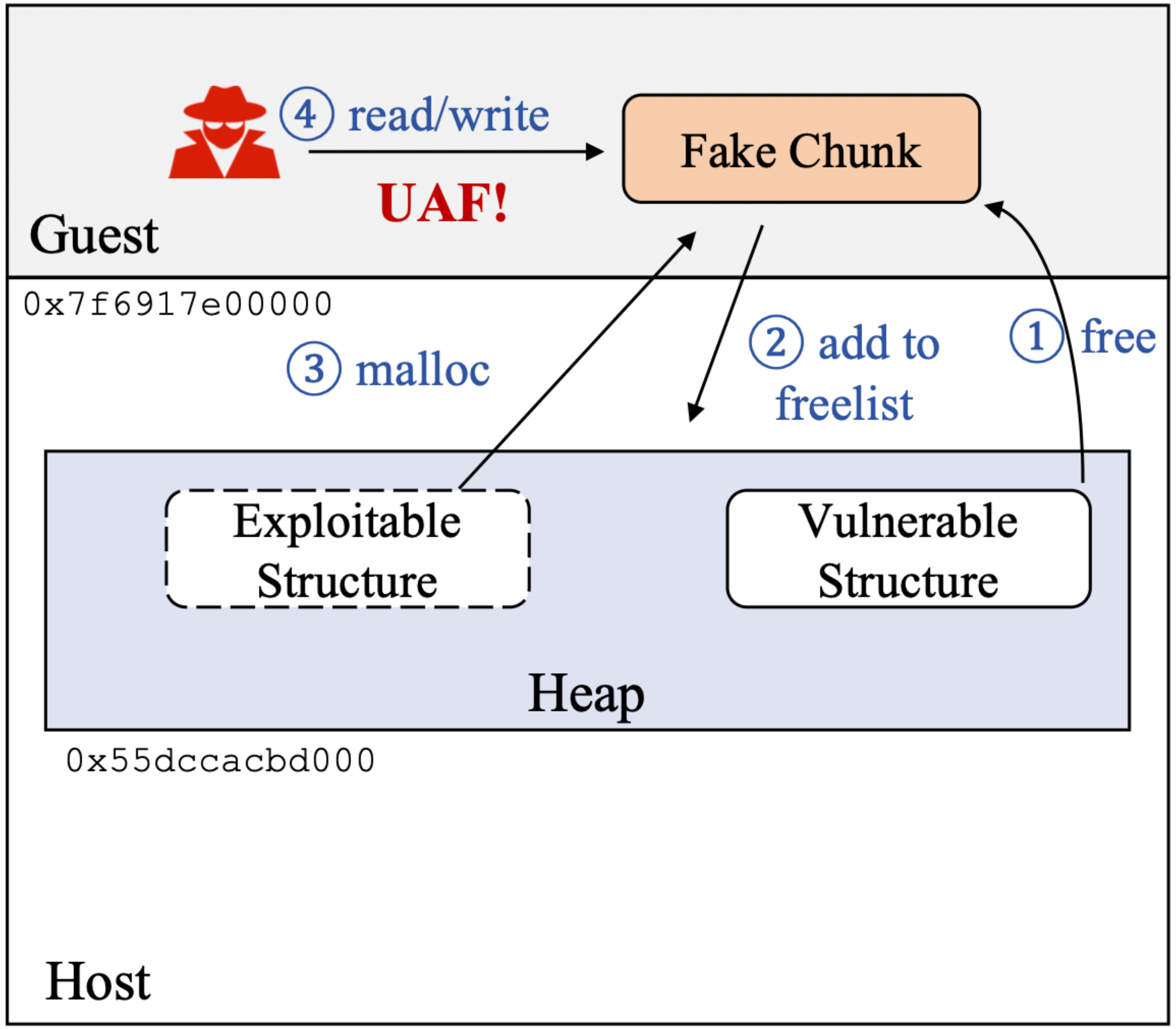
Both the stack and the heap provide extensive and repeatable opportunities for CDA redirection.

Evaluation: Exploit Practicality

	CVE id/Name	Device	Vulnerability Type	CDA Variants	Impact	Success
QEMU	CVE-2019-6778	slirp	Heap overflow	**O,C**	RCE	✓
	CVE-2019-14378	slirp	Heap overflow	**O,C**	RCE	✓
	CVE-2020-7039	slirp	Heap overflow	**O,C**	RCE	✓
	CVE-2020-14364	USB	OOB	**A,I**	RCE	✓
	CVE-2021-3682	USB redirector device	Mistake free	**O,C**	RCE	✓
	CVE-2021-3929	Nvme	UAF	**O,C**	RCE	✓
	CVE-2023-3180	virtio-crypto	Heap overflow	**A,I**	RCE	✓
	CVE-2023-6693	virtio-net	Stack overflow	**I**	Info leak	✓
	Scavenger	NVMe	Uninitialized free	**O,C**	RCE	✓
	Fixes: 1733eebb9e7	virtio-iommu	OOB read	**I**	Info leak	✓
	CVE-2024-3446	virtio-gpu	Double free	**O,C**	RCE	✓
	CVE-2024-8612	virtio-blk	OOB read	**I**	Info leak	✓
	Fixes: 62dbe54c	virtio-sound	Heap overflow	**A**	RCE	✓
VirtualBox	CVE-2020-2575	usb-ohci	Uninitialized heap	**A**	RCE	✓
	CVE-2020-2758	VHWA	UAF	**A,I**	RCE	✓

★ Successfully exploited **15 previously hard-to-exploit** vulnerabilities across QEMU and VirtualBox.

Case Study



CDA-Based exploitation of an uninitialized free in QEMU's NVMe

Possible Defense Mechanism

- **Memory Access Control**: Prevent host from accessing guest memory by default, similar to SMEP/SMAP in kernel space.

- **Gadget Reduction**: Eliminate raw guest-HVA pointers in host memory. Use handles, offsets, or opaque tokens.

Conclusion & Impact

√ Guest memory becomes a reusable exploitation substrate

√ First systematic characterization of Cross-Domain Attacks

√ Successful exploitation of 15 real-world vulnerabilities

√ CDA: https://github.com/HDU-SEC/cda

> “ We hope this work raises awareness about the risks of **implicit trust in guest memory** and motivates **stronger isolation mechanisms** in virtualization security.

Q&A

pgn@hdu.edu.cn