UniSan:
Proactive Kernel Memory Initialization to Eliminate Data Leakages

Kangjie Lu, Chengyu Song, Taesoo Kim, Wenke Lee

School of Computer Science,
Georgia Tech

Any Problem Here?

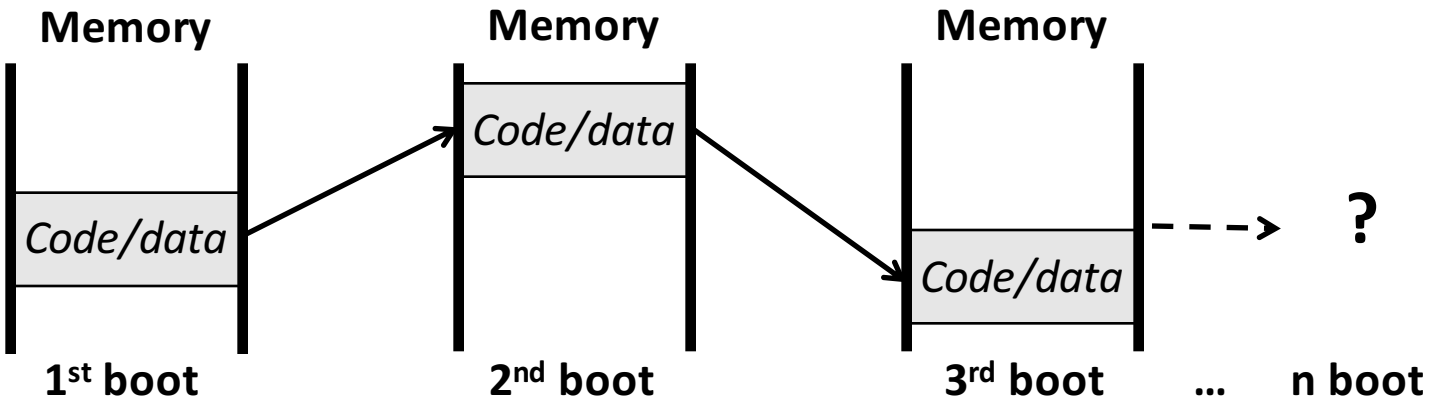
```
/* File: drivers/usb/core/devio.c */
/* define data structure “usbdevfs_connectinfo” */
struct usbdevfs_connectinfo {
    unsigned int devnum;
    unsigned char slow;
};

/* create and initialize object “ci” */
struct usbdevfs_connectinfo ci = {
    .devnum = ps->dev->devnum,
    .slow = ps->dev->speed == USB_SPEED_LOW
};
/* copy “ci” to user space */
copy_to_user(arg, &ci, sizeof(ci));
```

3-byte padding

Information leak!

Security Mechanisms in OS Kernels

kASLR: Randomizing the address of code/data



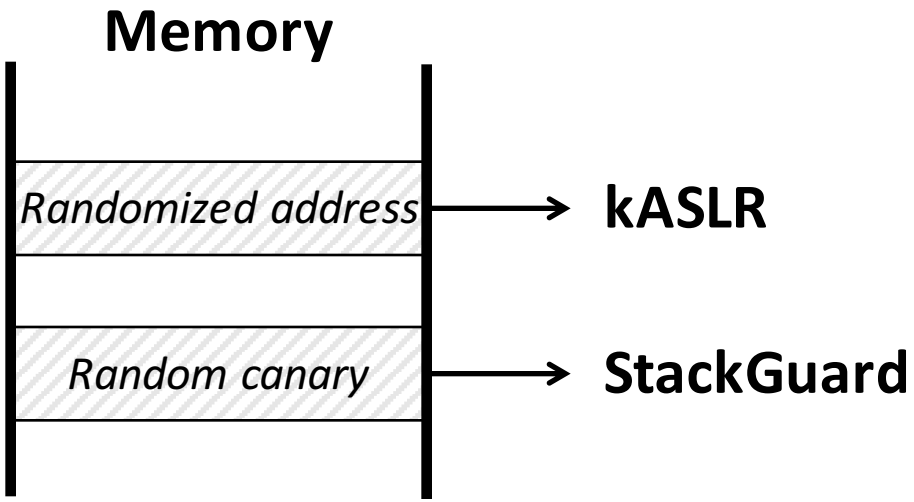
– Preventing code-reuse and privilege escalation attacks

StackGuard: Inserting random canary in stack

– Preventing stack corruption-based attacks

The Assumption of Effectiveness

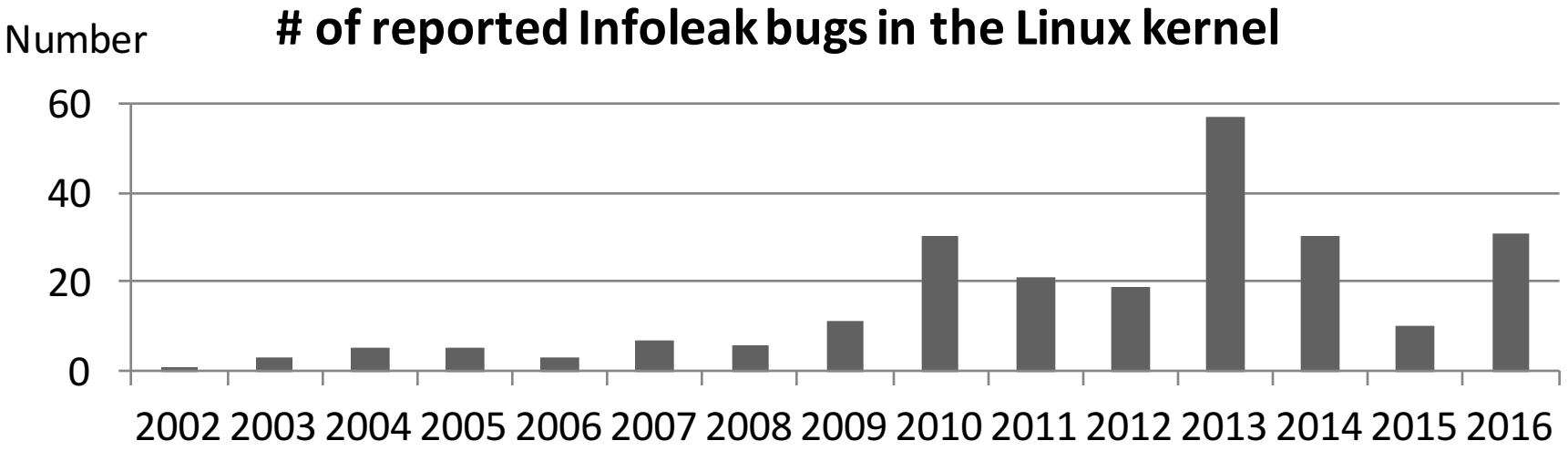
Assumption: No information leak



A single information leak renders these security mechanisms ineffective!

Infoleak in the OS (Linux) Kernel

According to the CVE database



These security mechanisms are often bypassed in reality

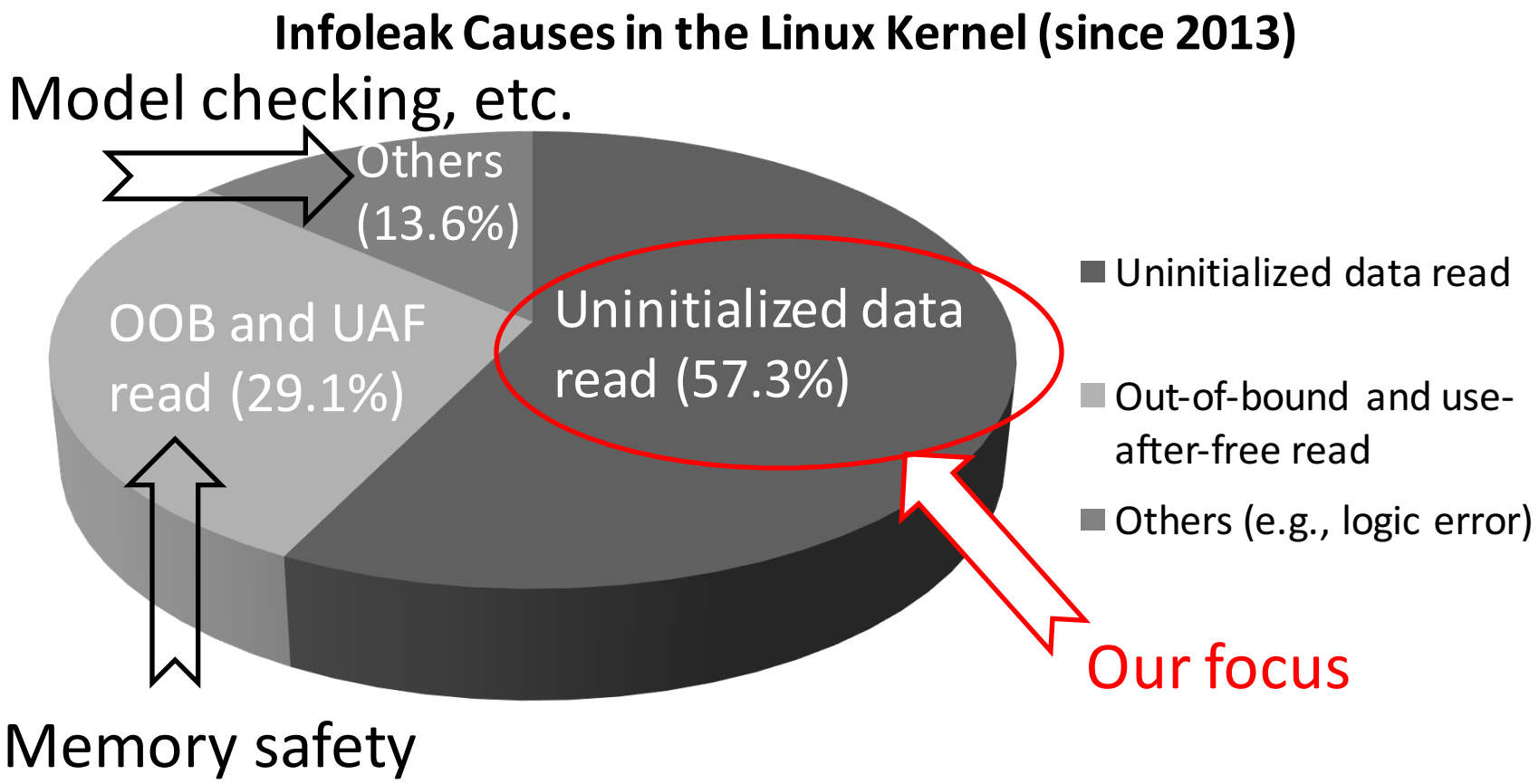
Sensitive data (e.g., cryptographic keys) can also be leaked.

Our research aims to eliminate information leaks in OS kernels

Causes of Infoleaks

- **Uninitialized data read**: Reading data before initialization, which may contain uncleared sensitive data
- **Out-of-bound read**: Reading across object boundaries
- **Use-after-free**: Using freed pointer/size that can be attacker controlled
- **Others**: Missing permission check, race condition

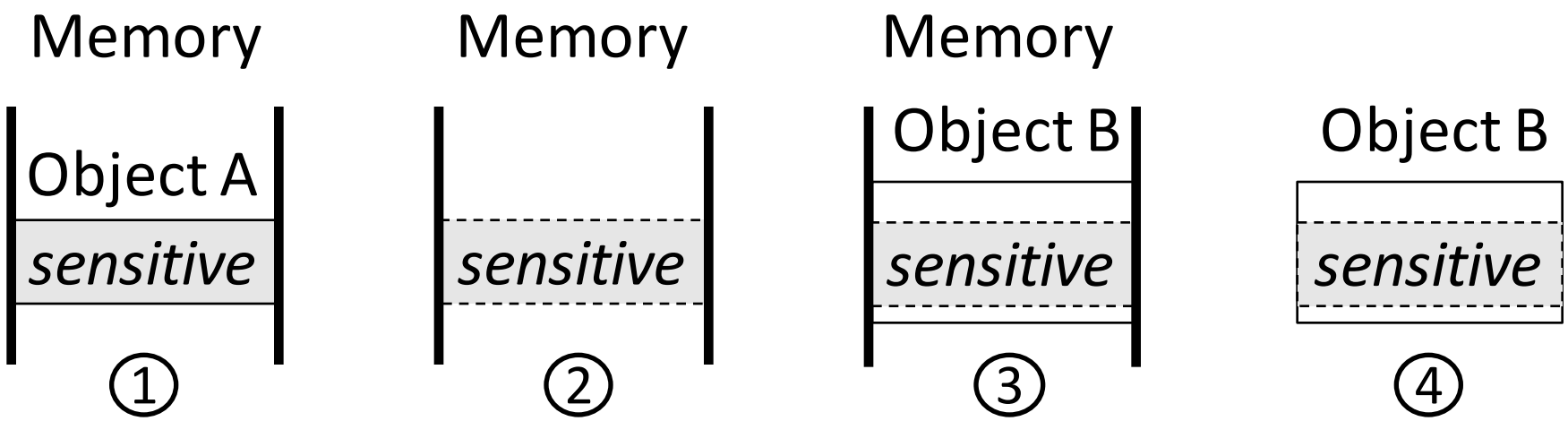
Causes of Infoleaks (cont.)



Similarly, Chen et al. [APSys'11] showed 76% infoleaks (Jan. 2010 - Mar. 2011) are caused by uninitialized data reads

From Uninitialized Data Read to Leak

1. Deallocated memory is not cleared by default.
2. Allocated memory is not initialized by default.
3. Reading the uninitialized memory -> **leak.**



User A allocates object A and writes "*sensitive*" in to it

User A deallocates object A; "*sensitive*" is not cleared

User B allocates object B without Initialization; "*sensitive*" kept

User B reads Object B; "*sensitive*" leaked!

Troublemaker: Developer

Missing element initialization: Blame the developer. ☺

Difficult to avoid, e.g.,

- Data structure definition and object initialization may be implemented by different developers

Troublemaker: Compiler

Data structure padding: A fundamental feature improving CPU efficiency

```
struct usbdevfs_connectinfo {
    unsigned int devnum;
    unsigned char slow;
    /* 3-bytes padding */
};
```

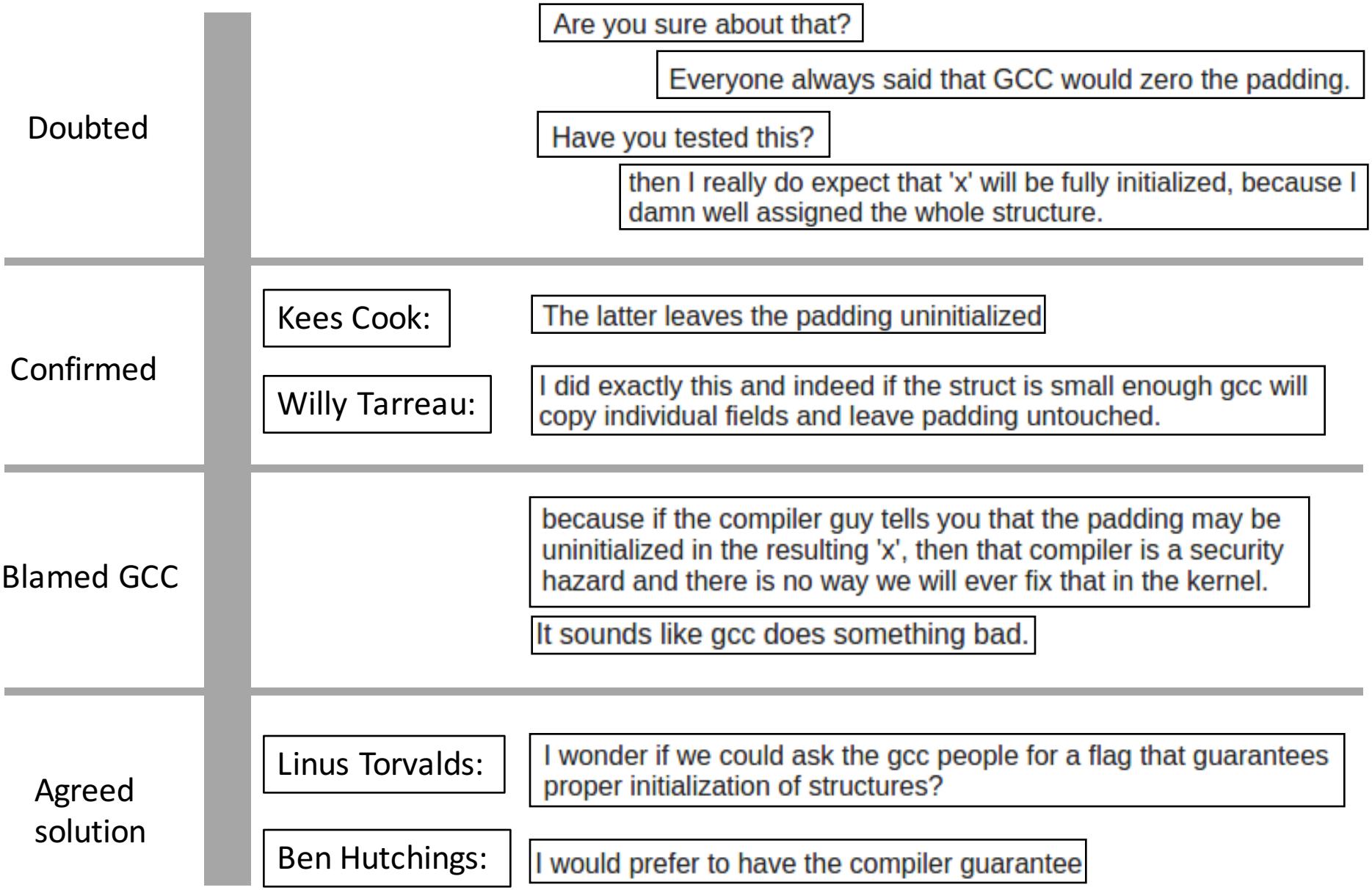
```
/* both fields (5 bytes) are initialized */
struct usbdevfs_connectinfo ci = {
    .devnum = ps->dev->devnum,
    .slow = ps->dev->speed ==
            USB_SPEED_LOW
};
/* leaking 3-byte uninitialized padding
    sizeof(ci) = 8 */
copy_to_user(arg, &ci, sizeof(ci));
```

C Specifications (C11)

Chapter §6.2.6.1/6

"When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values."

Responses from the Linux Community



Detecting/Preventing Uninitialized Data Leaks

The -*Wuninitialized* option of compilers?

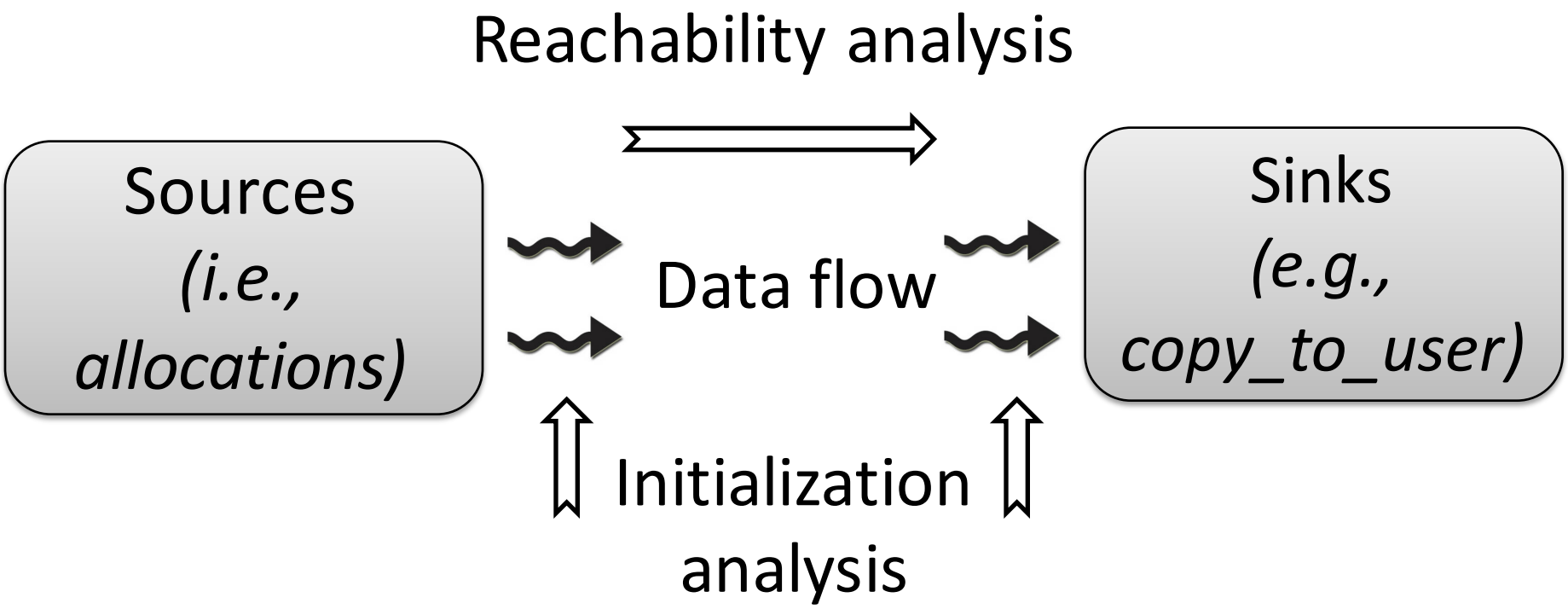
Simply initialize all allocations?

Our **UniSan** approach:

1) Conservatively identify unsafe allocations (i.e., with potential leaks) via static program analysis
2) Instrument the code to initialize only unsafe allocations

Detecting Unsafe Allocations

Integrating byte-level and flow-, context-, and field-sensitive reachability and initialization analyses



Main Challenges in UniSan

- Sink definition
  - General rules
- Global call-graph construction
  - Type analysis for indirect calls
- Byte-level tracking
  - Offset-based analysis, “GetElementPtr”

Be conservative!
Assume it is unsafe for special cases!

Instrumentation

Zero-initializations for unsafe allocations:

- Stack: Assigning zero or using *memset*
- Heap: Adding the __GFP_ZERO flag to *kmalloc*

Instrumentations are semantic preserving

- Robust
- Tolerant of false positives

Implementation

- Using LLVM
  - An analysis pass and an instrumentation pass
- Making kernels compilable with LLVM
  - Patches from the LLVMLinux project and Kenali [NDSS'16]
- Optimizing analysis
  - Modeling basic functions

How to use UniSan:

```
$ unisan @bitcode.list
```

Evaluation

Evaluation goals

- Accuracy in identifying unsafe allocations
- Effectiveness in preventing uninitialized data leaks
- The efficiency of the secured kernels

Platforms

- Latest mainline Linux kernel for x86_64
- Latest Android kernel for AArch64

Evaluation of Accuracy

Statistics of various numbers:

- Only **10%** of allocations are detected as unsafe.

Arch	Module	Alloca	Malloc	Unsafe Alloca	Unsafe Malloc	Percent
X86_64	2,152	17,878	2,929	1,493	386	**9.0%**
AArch64	2,030	15,628	3,023	1,485	451	**10.3%**

Evaluation of Effectiveness

Preventing known leaks:

- Selected 43 recent leaks with CVE#
- UniSan prevented all of them

Detecting unknown leaks

- With manual verification

Confirmed New Infoleaks (Selected)

File	Object	Leak Bytes	Cause	CVE
rtnetlink.c	map	4	Pad	CVE-2016-4486
devio.c	ci	3	Pad	CVE-2016-4482
af_llc.c	info	1	Pad	CVE-2016-4485
timer.c	tread	8	Pad	CVE-2016-4569
timer.c	r1	8	Pad	CVE-2016-4578
netlink...c	link_info	60	Dev.	CVE-2016-5243
media-device.c	u_ent	192	P&D	AndroidID-28616963
more…	more…	…	…	more…

Evaluation of Efficiency

Runtime overhead (geo-mean %)

Category	Benchmarks	Blind Mode (x86_64)	UniSan (x86_64)
System operations	LMBench	4.74%	1.36%
Server programs	ApacheBench	0.8%	<0.1%
User programs	SPEC Bench	1.92%	0.54%

Analyses took less **3** minutes.

Binary size increased < **0.5%.**

Limitations and Future Work

- Custom heap allocators
  - Require annotations
- Close-sourced modules
  - Not supported
- Other uninitialized uses, e.g., uninitialized pointer dereference
- GCC support (in progress)

Conclusions

- Information leaks are common in OS kernels.
- Uninitialized read is the dominant cause.
- Developers are not always to blame—compilers may also introduce security problems.
- UniSan eliminates all uninitialized data leaks.

Try UniSan:

https://github.com/sslab-gatech/unisan