HART: Hardware-assisted Modular Tracing on ARM

Yunlan Du[1,*], Zhenyu Ning[2,*], Jun Xu[3], Zhilong Wang[4], Yueh-Hsun Lin[5], Fengwei Zhang[2], Xinyu Xing[4], Bing Mao[1]

[1]Nanjing University
[2]Southern University of Science and Technology
[3]Stevens Institute of Technology
[4]Pennsylvania State University
[5]JD Silicon Valley R&D Center
[*]These authors contributed equally to this work

ESORICS, Sep 15, 2020

Outline

- Introduction
- Background
- HART: Hardware-Assisted Runtime Tracing framework
- HASAN: HART-based Address Sanitizer
- Evaluation
- Conclusion

Outline

- Introduction
- Background
- HART: Hardware-Assisted Runtime Tracing framework
- HASAN: HART-based Address Sanitizer
- Evaluation
- Conclusion

Introduction

The vulnerabilities in kernel modules have been a serious threat for the security of the Linux kernel.

- Caused by lacking of code correctness and testing rigorousness
- CVE patches to kernel drivers comprise roughly of 19% commits from 2005 to 2017 [1, 2].
- In 2017, 41% of 660 collected bugs in Android ecosystem came from kernel components most of which were device drivers [3].

Introduction

- To solve the problem, many solutions have been proposed

Approach Category		**Representative Works (in the Order of Time)**
Memory Debugger		Slub_debug [4], Kmemleak [5], Kmemcheck [6], KASAN [7]
Integrity Protection		KOP [8], HyperSafe [9], HUKO [10], KCoFI [11], DFI for kernel [12]
Kernel Isolation		Nooks [13], SUD [14], Livewire [15], SafeDrive [16], SecVisor [17]

Table: Existing kernel protection works.

Introduction

- To solve the problem, many solutions have been proposed
- But, the problem is far from solved

Approach Category	**Binary -support**	**Non -intrusive**	**Low overhead**	**Representative Works (in the Order of Time)**
Memory Debugger	✗	✗	✗	Slub_debug [4], Kmemleak [5], Kmemcheck [6], KASAN [7]
Integrity Protection	✗	✗	✱	KOP [8], HyperSafe [9], HUKO [10], KCoFI [11], DFI for kernel [12]
Kernel Isolation	✗	✗	✱	Nooks [13], SUD [14], Livewire [15], SafeDrive [16], SecVisor [17]

Table: Existing kernel protection works.
(✓ = yes, ✗ = no, ✱ = partially supported.)

Introduction

Motivation: Build a high-performance tracing framework for unmodified kernel modules without module source code

Outline

- Introduction
- Background
- HART: Hardware-Assisted Runtime Tracing framework
- HASAN: HART-based Address Sanitizer
- Evaluation
- Conclusion

Embedded Trace Macrocell

Embedded Trace Macrocell (ETM) is a hardware component on Arm processors. It is able to tracing the instruction execution and memory access with negligible overhead.

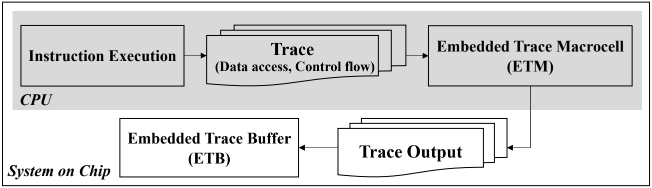


Figure: A general hardware model of ETM.

Outline

- Introduction
- Background
- HART: Hardware-Assisted Runtime Tracing framework
- HASAN: HART-based Address Sanitizer
- Evaluation
- Conclusion

Hardware-Assisted Runtime Tracing framework

- HART, a **H**ardware-**A**sssited **R**untime **T**racing framework

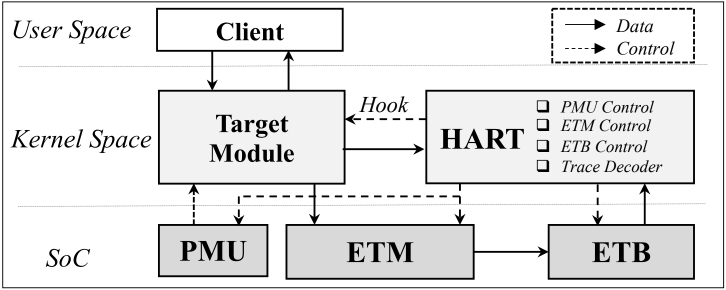


Figure: Architecture of HART framework.

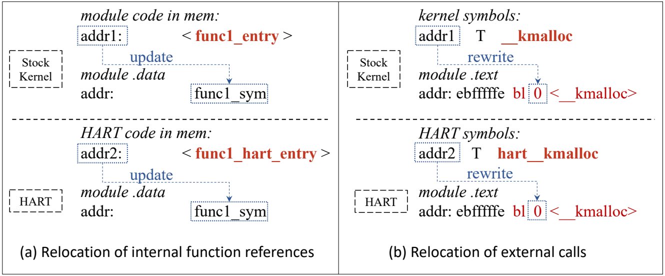
Selective Tracing

Challenge 1: Selective Tracing

- As a hardware component, ETM is lacking of OS semantics
  - Filters in ETM are limited
  - Hard to identify the trace of target module from the output
- Size of trace buffer is limited
  - Tracing the entire execution in the processor leads to frequent overflow
  - To trace the other components in the system is a waste of resource

Selective Tracing

Solution: Selective Tracing via hooking and wrapping



(a) Relocation of internal function references

(b) Relocation of external calls

- Hook the entrances and exits during the module loading stage
  - Achieved by callbacks registered via trace-point, without intrusion to the kernel
- Replace entrances and exits with wrappers at relocation stage
  - Including code points in `.data` and `.text` segments

Continuous Tracing

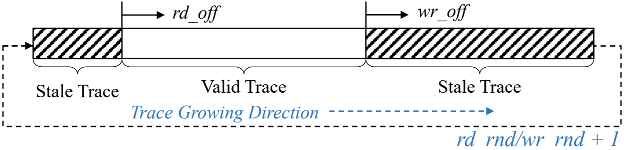
Challenge 2: Continuous Tracing

- The size of the trace buffer in SoCs are limited
  - According to our observation, normally 4k trace buffer is implemented
  - Could be fully occupied in milliseconds or seconds
- The overflow of the trace buffer leads to losing of trace
  - The trace buffer is a ring buffer
  - Older trace data will be overridden after overflow

Continuous Tracing

Solution: Continuous Tracing via timely interrupts

- Leverage PMU to issue an interrupt before overflow
    - In general, at most 6 byte trace data per instruction
    - We make 670 instructions as the threshold, and issue an interrupt after every 670 instructions are executed
- During the interrupt, validate and extract the trace with careful designed algorithm



High-performance Tracing

Challenge 3: High-performance Tracing

- The overhead of ETM tracing is negligible
- But, it takes performance to handle the trace
  - Extracting data from the trace buffer
  - Decoding the trace data

High-performance Tracing

Solution: High-performance Tracing via elastic decoding

- A dedicated decoding thread
- Yielding CPU based on the workload of the decoding thread
  - Calculating extracted data size
  - To yield according to the data size

Outline

- Introduction
- Background
- HART: Hardware-Assisted Runtime Tracing framework
- HASAN: HART-based Address Sanitizer
- Evaluation
- Conclusion

HART-based Address Sanitizer

HASAN: a HART-based address sanitizer, reusing the scheme of AddressSanitizer [18]

- Redzones for out of bound detection
    - Wrapping objects with redzones
    - Accessing the redzones leads to fault
- Shadow memory for memory tags
    - `0xbf000000` to `0xffffffff` as kernel space in our system
    - Allocate 130M continuous virtual space as shadow memory

HART-based Address Sanitizer

HASAN: a HART-based address sanitizer.

- *With module source code:*
    - Both HASAN and KASAN can achieve heap & stack protection
- *Without module source code:*
    - HASAN achieves heap protection
    - KASAN would not work at all

HART-based Address Sanitizer

Heap protection without module source code

- Achieved by hooking the `slab` interfaces for memory management

Category	**Allocation**	**De-allocation**
Kmem_cache	`kmem_cache_alloc` `kmem_cache_create`	`kmem_cache_free` `kmem_cache_destroy`
Kmalloc	`kmalloc` `krealloc` `kzalloc` `kcalloc`	`kfree`
Page operations	`alloc_pages` `__get_free_pages`	`free_pages` `__free_pages`

Table: Memory management interfaces HASAN hooked.

Outline

- Introduction
- Background
- HART: Hardware-Assisted Runtime Tracing framework
- HASAN: HART-based Address Sanitizer
- Evaluation
- Conclusion

Evaluation

Experiment setup:

- Freescale i.MX53 Quick Start Board
- Raspberry Pi 3+ for KASAN
  - We implement HASAN in 32-bit i.MX53 QSB, but KASAN only support 64-bit systems
- `lmbench`, and 6 widely-used kernel modules with standard benchmarks

Overhead to the main kernel

Func.	Setting	Native	KASAN	Overhead
Processes (ms)	stat	3.08	16.4	5.3
	open clos	8.33	36.7	4.4
	sig hndl	6.06	20.4	3.4
	fork proc	472	1940	4.1
Local Comm. latency (ms)	Pipe	18.9	45.8	2.4
	AF UNIX	26.6	97.9	3.7
	UDP	41.4	127.6	3.1
	TCP	53.4	176.4	3.3
File & VM system latency (ms)	0K File Create	44.0	136.1	3.1
	0K File Delete	35.2	227.1	6.5
	10K File Create	99.9	370.2	3.7
	10K File Delete	64.2	204.7	3.2
	Mmap Latency	188000	385000	2.0
	Prot Fault	0.5	0.5	1.0
	Page Fault	1.5	2.3	1.5
	100fd selct	6.6	13.7	2.1

Table: Performance evaluation on KASAN with `lmbench`. HART and HASAN introduce no overhead to the main kernel, so the results are omitted here.

Performance evaluation

Module		Benchmark		Result			
Type	Name	Name	Setting	Native img +		KASAN img +	
				HART module	HASAN module	Native module	KASAN module
Network	HSTCP [19]	iperf [20]	Local Comm.	1.00	1.00	0.29	0.28
	TCPW [21]	iperf [20]	Local Comm.	0.92	0.91	0.28	0.28
	H-TCP [22]	iperf [20]	Local Comm.	0.94	0.94	0.26	0.25
File System	HFS+ [23]	IOZONE [24]	Wr/fs=4048K/reclen=64	1.00	1.00	0.96	0.95
			Wr/fs=4048K/reclen=512	0.88	0.87	0.96	0.94
			Rd/fs=4048K/reclen=64	0.92	0.89	0.98	0.92
			Rd/fs=4048K/reclen=512	0.90	0.89	0.99	0.99
	UDF [25]	IOZONE [24]	Wr/fs=4048K/reclen=64	0.95	0.93	0.99	0.97
			Wr/fs=4048K/reclen=512	0.97	0.97	1.00	0.92
			Rd/fs=4048K/reclen=64	0.98	0.97	0.99	0.98
			Rd/fs=4048K/reclen=512	0.97	0.96	1.00	0.98
Driver	USB_STORAGE[26]	dd [27]	Wr/bs=1M/count=1024	1.00	1.00	1.00	0.43
			Wr/bs=4M/count=256	1.00	1.00	0.99	0.43
			Rd/bs=1M/count=1024	0.99	0.99	0.99	0.75
			Rd/bs=4M/count=256	1.00	1.00	1.00	0.76
Avg.	-	-	-	0.95	0.94	0.85	0.72

Table: Performance evaluation with kernel modules and benchmarks.

Tracing evaluation

Module		**Retrieving times**	
Type	Name	HART	HASAN
Network	HSTCP	4243	3964
	TCP-W	3728	3584
	H-TCP	3577	3595
File	HFS+	30505	30278
Driver	USB_STORAGE	9316	9325

Module		**Max size(Byte)**		**Min size(Byte)**		**Average size(Byte)**		**Full ETB**	
Type	Name	HART	HASAN	HART	HASAN	HART	HASAN	HART	HASAN
Network	HSTCP	1100	1196	20	20	988	1056	0	0
	TCP-W	1460	1456	20	20	1128	1088	0	0
	H-TCP	1292	1304	20	20	1176	1168	0	0
File System	HFS+	1652	1756	20	20	144	148	0	0
	UDF	2424	2848	20	20	240	232	0	0
Driver	USB_STORAGE	1544	1692	20	20	448	448	0	0

Table: Tracing evaluation of HART and HASAN.

Effectiveness evaluation

Vulnerability		**Detection**		
CVE-ID	**Type**	**PoC**	**HASAN**	**KASAN**
CVE-2016-0728	Use-after-free	REFCOUNT overflow [28]	Y	Y
CVE-2016-6187	Out-of-bound	Heap off-by-one [29]	Y	Y
CVE-2017-7184	Out-of-bound	xfrm_replay_verify_len [30]	Y	Y
CVE-2017-8824	Use-after-free	dccp_disconnect [31]	Y	Y
CVE-2017-2636	Double-free	n_hdlc [32]	Y	Y
CVE-2018-12929	Use-after-free	ntfs_read_locked_inode [33]	Y	Y

Table: Effectiveness evaluation on HASAN.

Outline

- Introduction
- Background
- HART: Hardware-Assisted Runtime Tracing framework
- HASAN: HART-based Address Sanitizer
- Evaluation
- Conclusion

Conclusion

- We present HART, a hardware-based high-performance tracing framework specially for kernel modules

- Based on the HART, we build a modular security solution, HASAN, to effectively detect memory corruptions without requiring the source code of the module

- The evaluation result shows that HASAN can achieve the detection with only 5%-6% performance overhead, which is significantly superior to the state-of-the-art solution KASAN

References I

[1] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, “Linux kernel vulnerabilities: State-of-the-art defenses and open problems,” in Proceedings of the Second Asia-Pacific Workshop on Systems. ACM, 2011, p. 5.

[2] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, “Dr. checker: A soundy analysis for linux kernel drivers,” in 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, 2017, pp. 1007–1024.

[3] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velásquez, “An empirical study on android-related vulnerabilities,” in Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on. IEEE, 2017, pp. 2–13.

[4] “slub,” https://www.kernel.org/doc/Documentation/vm/slub.txt, 2017.

[5] “Kmemleak,” https://www.kernel.org/doc/html/v4.14/dev-tools/kmemleak.html, 2019.

[6] “Getting started with kmemcheck – the linux kernel documentation,” https://www.kernel.org/doc/html/v4.14/dev-tools/kmemcheck.html, 2019.

[7] “Home google/kasan wiki,” https://github.com/google/kasan/wiki, 2018.

[8] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, “Mapping kernel objects to enable systematic integrity checking,” in Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009, pp. 555–565.

[9] Z. Wang and X. Jiang, “Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity,” in Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 2010, pp. 380–395.

[10] X. Xiong, D. Tian, P. Liu et al., “Practical protection of kernel integrity for commodity os from untrusted extensions,” in NDSS, vol. 11, 2011.

[11] J. Criswell, N. Dautenhahn, and V. Adve, “Kcofi: Complete control-flow integrity for commodity operating system kernels,” in Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 2014, pp. 292–307.

References II

[12] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, “Enforcing kernel security invariants with data flow integrity.” in NDSS, 2016.

[13] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, “Nooks: An architecture for reliable device drivers,” in Proceedings of the 10th workshop on ACM SIGOPS European workshop. ACM, 2002, pp. 102–107.

[14] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, “Fast byte-granularity software fault isolation,” in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009, pp. 45–58.

[15] S. Boyd-Wickizer and N. Zeldovich, “Tolerating malicious device drivers in linux.” in USENIX Annual Technical Conference. Boston, 2010.

[16] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, “Safedrive: Safe and recoverable extensions using language-based techniques,” in Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006, pp. 45–60.

[17] A. Seshadri, M. Luk, N. Qu, and A. Perrig, “Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses,” in ACM SIGOPS Operating Systems Review. ACM, 2007, pp. 335–350.

[18] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Addresssanitizer: A fast address sanity checker.” in USENIX Annual Technical Conference, 2012, pp. 309–318.

[19] S. Floyd, “Highspeed tcp for large congestion windows,” https://tools.ietf.org/html/rfc3649, 2003.

[20] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, K. Prabhu, and etc., “iperf - the ultimate speed test tool for tcp, udp and sctp,” https://iperf.fr/, 2018.

[21] “Tcp westwood+ congestion control,” https://tools.ietf.org/html/rfc3649, 2003.

[22] “H-tcp - congestion control for high delay-bandwidth product networks,” http://www.hamilton.ie/net/htcp.htm, 2019.

References III

[23] “Hfs plus,” https://www.forensicswiki.org/wiki/HFS%2B, 2019.

[24] C. Don, C. Capps, D. Sawyer, J. Lohr, G. Dowding, and etc., “Iozone filesystem benchmark,” http://www.iozone.org/, 2016.

[25] “Universal disk format,” https://docs.oracle.com/cd/E19683-01/806-4073/fsoverview-8/index.html, 2019.

[26] “Config_usb_storage: Usb mass storage suppor,” https://cateee.net/lkddb/web-lkddb/USB_STORAGE.html, 2019.

[27] P. Rubin, D. MacKenzie, and S. Kemp, “dd - convert and copy a file,” http://man7.org/linux/man-pages/man1/dd.1.html, 2019.

[28] P. P. Team, “Refcount overflow exploit,” https://github.com/SecWiki/linux-kernel-exploits/blob/master/2016/CVE-2016-0728/cve-2016-0728.c, 2017.

[29] V. Nikolenko, “Heap off-by-one poc,” http://cyseclabs.com/exploits/matreshka.c, 2016.

[30] snorez, “Exploit of cve-2017-7184,” https://raw.githubusercontent.com/snorez/exploits/master/cve-2017-7184/exp.c, 2017.

[31] M. Ghannam, “Cve-2017-8824 linux: use-after-free in dccp code,” https://www.openwall.com/lists/oss-security/2017/12/05/1, 2017.

[32] A. Popov, “Cve-2017-2636: exploit the race condition in the n_hdlc linux kernel driver bypassing smep,” https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html, 2017.

[33] S. Schumilo, “Multiple memory corruption issues in ntfs.ko (linux 4.15.0-15.16),” https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1763403, 2018.

Thank you!

Questions?

{duyunlan}@smail.nju.edu.cn