Fast memory debugger for large software projects

Yury Gribov, Maria Guseva, Andrey Ryabinin, JaeOok Kwon, SeungHoon Lee, HakBong Lee, ChungKi Woo

***Abstract*— C/C++ programs often suffer from memory corruption bugs. Over the years, numerous tools were developed to help with their detection. A recent addition is AddressSanitizer (ASan) - an extraordinarily fast runtime checker with a good coverage of various types of bugs.**

This paper describes our experience in integration of ASan technology into large-scale software products: Tizen distribution and Linux kernel. The tool has already found around a hundred of serious memory bugs in various Tizen applications and in mainline Linux kernel.

***Keywords*—runtime memory checker, AddressSanitizer, KernelAddressSanitizer.**

I. Introduction

Memory corruption is an error which occurs when application unintentionally reads or writes data past the bounds of proper memory region. Typical examples are buffer overflows and use-after-free errors. Examples of such errors are given in Fig. 1.

```
int x[10];
for (int i = 0; i <= 10; ++i)
  x[i] = 0;  // Buffer overflow @ i == 10

char *p = malloc (1);
free (p);
p[0] = 0;  // Use-after-free
```

Fig. 1. Examples of memory errors (buffer overflow and use-after-free).

Various approaches are used to detect memory errors at early development stages, including code reviews, static analysis, managed languages, etc. [1]. One important class of tools is runtime memory checkers ("memory debuggers") which combine high precision with cheap integration costs.

A relatively new addition to runtime checkers family is AddressSanitizer (or shortly ASan) [2]. ASan is unique in that it has only 2x performance overhead and consumes 10% of memory which is unparalleled by prior technology (Valgrind, ASan's most direct and popular competitor, incurs an overhead of 30x! [3]). ASan fully supports multi-threading which is important for high-performance server applications.

Manuscript received August 21, 2015.

Yury Gribov (y.gribov@samsung.com), Maria Guseva (m.guseva@samsung.com) and Andrey Ryabinin (a.ryabinin@samsung.com) are with Samsung R&D Institute Russia, 12-1 Dvintsev st., Moscow, Russia, 127018

JaeOok Kwon, SeungHoon Lee, HakBong Lee and ChungKi Woo are with Samsung Electronics Company

ASan detects many classes of bugs. Notably it has been proven to find the infamous Heartbleed exploit in OpenSSL [4]. Current version of ASan is capable of detecting buffer overflows (in stack, heap and static memory), use-after-free and use-after-return, initialization order fiasco, memory leaks, trivial heap errors (double free, free-delete mismatch, etc.) and some other errors (memcpy parameter overlap, etc.). Summary comparison of ASan and other memory tools see in Table I.

Table I. Comparison of memory tools.

Feature \ Tool	Valgrind	Guard page tools (DUMA, Efence, etc.)	ASan
Technology	Dynamic instrumentation	Dynamic instrumentation	Compile-time instrumentation
Supported platforms	Linux, Mac	All	All
Overhead	20x	1x	2x
Multithreading support	No	Yes	Yes
Heap overflow	Yes	Yes	Yes
Global overflow	No	No	Yes
Stack overflow	No	No	Yes
Use-after-free	Yes	Yes	Yes
Use-after-return	No	No	Yes
Memory leaks	Yes	Yes	Yes
ODR, init order violation	No	No	Yes

AddressSanitizer is based on a classic shadow memory approach to memory error detection which is also used in tools like Valgrind or kmemcheck. Shadow memory is a special memory region in program's memory which holds information about state of user's data i.e. which memory locations are unsafe to access and why [5]. ASan is using 8-to-1 encoding (see Fig. 2) i.e. each 8-byte program word is mapped to 1 byte of shadow memory. This encoding allows efficient code generation particularly on 64-bit platforms (Fig. 3).

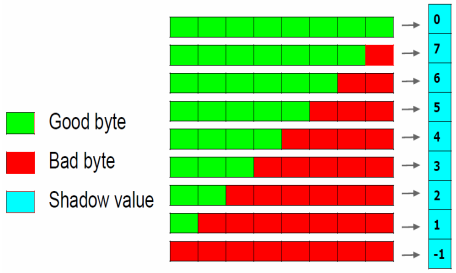

Fig. 2. ASan memory encoding.

```
// Original code
Type val = *address;

//Instrumented code
char *shadow_address = 0x20000000 + (address >> 3)
char shadow_val = *shadow_address;
char last_byte = (address & 7) + sizeof(Type) - 1;
if (*shadow_val && last_byte >= *shadow_val)
  ReportError();
Type val = *address;
```

Fig. 3. ASan instrumentation.

Contrary to Valgrind and Purify, ASan uses compile-time instrumentation to query and check shadow memory on each scalar memory access. This allows for much better optimization and removal of redundant computation resulting in better performance. ASan runtime library also intercepts many Glibc memory functions (*memset*, *strcpy*, etc.) to catch invalid memory accesses in them.

To detect memory errors, AddressSanitizer pads various program objects (heap, stack and global variables) with poisoned (i.e. marked as inaccessible in shadow memory) redzones. Buffer overflow would cause program to access poisoned region and trigger runtime fault with a helpful error message. Poisoning/un-poisoning is done

- for global variables – at program startup (via special hooks inserted by compiler)
- for stack variables – in function prologue/epilogue (via special instrumentation code inserted by compiler)
- for heap variables – in malloc/free and new/delete interceptors (located in ASan runtime library)

An error message typically includes type and context of faulty memory access and a backtrace. Example (truncated) report is shown in Fig. 4.

```
$ ./a.out
=================================================
==11083==ERROR: AddressSanitizer: heap-buffer-overflow
on address 0x60200000eff1 at pc 0x4007a8 bp 0x7ffd53cfb200
sp 0x7ffd53cfb1f8
WRITE of size 1 at 0x60200000eff1 thread T0
    #0 0x4007a7 in main /home/ygribov/tmp.c:3
    #1 0x7fc0b2ff9ec4 in __libc_start_main (/lib/x86_64-
linux-gnu/libc.so.6+0x21ec4)
    #2 0x400688 (/home/ygribov/a.out+0x400688)

0x60200000eff1 is located 0 bytes to the right of 1-byte
region [0x60200000eff0,0x60200000eff1)
allocated by thread T0 here:
    #0 0x7fc0b33f17df in __interceptor_malloc
(/usr/lib/x86_64-linux-gnu/libasan.so.1+0x547df)
    #1 0x400767 in main /home/ygribov/tmp.c:2
    #2 0x7fc0b2ff9ec4 in __libc_start_main (/lib/x86_64-
linux-gnu/libc.so.6+0x21ec4)

SUMMARY: AddressSanitizer: heap-buffer-overflow
/home/ygribov/tmp.c:3 main
==11083==ABORTING
```

Fig. 4. Example ASan report.

II. Large-Scale Projects Sanitizing

AddressSanitizer relies on standard and portable mechanisms like compiler flags, runtime interception of library functions and runtime tuning via environment variables. They work well for isolated software packages but may pose challenges when applied to a large software project like complete Linux distribution.

Several months ago we have successfully applied AddressSanitizer to ARMv7-based embedded system with Tizen software stack. Tizen is a Linux distribution aimed at consumer electronics devices (mobile phones, TVs, IVI, etc.) [6]. It is a typical example of modern software platform and we thus believe that our experience would be helpful for maintainers of other distributions (like Ubuntu or Android) who consider using ASan in their work.

Below we describe challenges met during sanitizing Tizen and how they were solved.

A. Integration

During ASan integration to Tizen we generally found that instead of doing things “properly” by modifying the platform build system core or package build scripts to match our requirements, it was much more efficient to work around arising problems.

Such basic task as modifying compiler flags for several thousands of packages in a scalable way may be non-trivial because each package may modify or override compiler flags in unique way. Thus modification of default compiler flags was achieved by a crude compiler wrapper script (see Fig. 5) which never failed us since then. In addition to forced enablement of ASan, we also disabled common symbols and Glibc fortification as both cause ASan to miss important classes of bugs (erroneous accesses to global variables or via standard memory functions like *memcpy*).

```
#!/bin/sh
# Use readlink in order to follow symlinks if any
REAL=$(readlink -f $0)-real
if ! echo "$@" | grep -q '__KERNEL__\|-nostdlib'; then
  $REAL "$@" -fsanitize=address -fno-common -
U_FORTIFY_SOURCE
else
  $REAL "$@"
fi
```

Fig. 5. Compiler wrapper for enabling ASan.

Once integration to build system has been finished and we have successfully rebuilt most part (actually 99.5%) of distribution with ASan, we were finally able to run the system. However in runtime we initially faced another issues like false error messages. We discovered their cause in several ARM-specific bugs on compiler side. Once we fixed them ASan proved to be extremely robust.

We also added some minor target-specific modification. E.g. in our case sanitized executables ran pretty early during system boot when proc partition (required by ASan to determine process memory layout, etc.) was not available. To address this, we updated ASan initialization code to mount /proc if necessary.

B. Instrumentation Overhead

Next to pure integration issues stands instrumentation overhead. Even though ASan is much more efficient than Valgrind, it provokes users to apply much more aggressively in new contexts (e.g. analyze full system under ASan). CPU overhead of 2x-3x is typically acceptable as it only results in moderate increase of QA time. Memory overhead is much more important – it may be unbearable for mobile devices with their limited amounts of RAM.

After initial experiments we quickly ran into problems with increased memory consumption. Our target devices were designed with particular usage scenarios in mind so amount of available RAM was limited and there was no secondary storage for swap. We attacked this problem from

different angles.

First of all, we used available ASan runtime options to reduce memory to minimum. ASan customization is done at runtime through environment variable ASAN_OPTIONS. We updated Tizen initialization scripts (/sbin/init.wrapper) to set the necessary options for all system processes:

- malloc_context_size=2,fast_unwind_on_malloc=0 (cap backtrace collection for malloc which is otherwise too slow and memory-heavy on most platforms)
- quarantine_size=$((1<<15)) (reduce size of quarantine to 32K to reduce memory consumption)
- start_deactivated=1 (do not enable ASan in non-sanitized programs)

In addition to above tweaks, we also enabled large swap on zram device [7][1].

An alternative solution to memory overhead (which was quickly adopted by users) is to split full distribution to smaller parts and concentrate on most critical parts first (e.g. set uid daemons or apps which access private user data). By applying ASan to a subset of distribution at a time we could arbitrarily trim memory overhead at the cost of increased QA time (as QA tests will now have to be run N times, once per each chunk). Unfortunately such partitioning may in practice cause false negatives[2]. For example if we apply ASan to an executable which links against library which was not sanitized, then errors inside library code will go undetected. This happens due to nature of ASan's instrumentation which requires that all source code (i.e. executable and all dependent libraries) is sanitized to achieve 100% error coverage. We are currently trying to automatically determine the minimal subset of distribution that would include all packages selected by user and all their dependencies (direct and transitive) required to detect all possible errors.

We also reduced consumption of virtual memory by removing kernel area image from shadow memory region and trimming too aggressive memory allocation in ASan upstream (see Fig. 6).

Resource	Improvement
Code size	25%
Virtual memory	30%
Performance	15%

Fig. 6. Achieved ASan overhead reduction.

Originally we didn't pay much attention to CPU overhead but once users started to use ASan more, we ran into limitations for high-performance workloads. On ARMv7 cores we were able to obtain a 25% code size reduction and a ~15% performance improvement on high loads by carefully tuning ASan instrumentation to our 32-bit ARM cores (we used ARM's dominated conditional comparisons). Example of instrumentations before and after our optimization is given in Fig. 7.

[1] We also plan to experiment with swapping to USB and network disks.
[2] I.e. some errors may be missed.

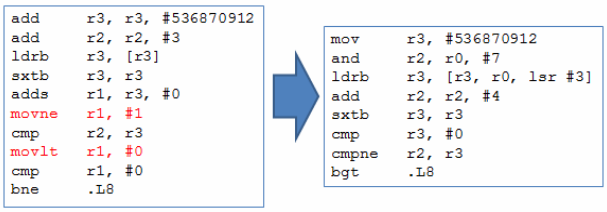

Fig. 7. Optimized instrumentation code.

C. *Other Limitations*

ASan's internal complexity is another source of issues. Asan code has many non-obvious limitations, loosely described (if at all) in documentation, mailing lists or even code comments. These should be carefully studied to ensure that important errors don't go undetected.

For example an important and non-obvious source of ASan limitations are custom memory allocators used by many important packages e.g. OpenSSL or Glib2 [9]. By design, without additional assistance from user ASan can only detect errors in dynamic memory allocated via standard malloc/free or new/delete allocators. Any custom memory handling (e.g. simple free list on top of *mmap*) is thus unknown to ASan and most use-after-free or buffer overflow errors there will go undetected. Some libraries provide means to disable custom allocators (e.g. G_SLICE=always_malloc setting in Glib2) but for the most part this work has to be done by Tizen application developers who are interested in expanding coverage of their code.

Finally, some ASan's design choices may further complicate its usage. The most unpleasant one is the decision to abort execution after single error detection. This approach aggressively motivates developers to fix bugs in their code but at the same time significantly limits number of errors that can be detected during a single QA run. Given that asking developers to fix code, rebuilding and reinstalling updated firmware may take anything from minutes to days, this may significantly increase QA cycle.

To avoid this we enabled new ASan runtime flag "keep_going" telling to continue execution after reporting an error instead of aborting. This allowed us to significantly increase our error detection rate and thus reduce integration costs. Looking back, we can confirm that lack of this feature would have significantly complicated adoption of ASan for QA. This flag is currently not available in mainline ASan (mainly due to maintainer's opposition) but we'd like to commit it in future.

D. *Ideas For Improving Test Coverage*

Once ASan was adopted and developers started to use it on a regular basis, we found that number of error detections has quickly diminished. This of course didn't mean that our software became bug-free but rather that coverage of our existing QA test suites was too narrow and they began to limit ASan's ability to detect errors. We conclude that dynamic checkers like ASan heavily depend on existence of aggressive and evolving test systems.

To detect more errors, we are currently exploring ways to increase our test coverage. One obvious approach is usage of fuzzing tools which have become very popular in recent years [8]. In addition to fuzzing, we also plan to strengthen

existing ASan checks by implementing more Glibc interceptors (e.g. bcopy, strrchr, etc.) and to experiment with more aggressive (and memory-hungry) ASan checks like *detect_stack_use_after_return* and *check_initialization_order*. We'd also like to extend ASan to cover less trivial constructs like *va_args* and thread-local storage.

III. ASAN SOLUTION FOR LINUX KERNEL: KASAN

In addition to Tizen, we also applied ASan to verification of Linux kernel. The solution named *KernelAddressSanitizer* (KASan) has been proposed by one of the authors to kernel community and successfully merged to mainline in version 4.0 [10].

Compiler instrumentation for kernel case remains practically unchanged but runtime support has to be rewritten from scratch. Checking of global and stack variables is more or less similar but dynamic memory allocation is quite different. For one thing, kernel does not use standard malloc/free interface but rather several different allocators meant for different purposes:

- Slab allocators (SLAB, SLOB, SLUB)
- Vmalloc
- Per-CPU variables
- Page allocator
- Memory pools

For now we have only added support for SLUB allocator as this seems to be the most popular implementation of Slab for now.

Initially, all objects allocated on SLUB-page are poisoned by KASan. Later, when SLUB-object is allocated, memory containing the object body is un-poisoned and the rest (e.g. SLUB-metadata) is marked poisoned (Fig. 8).

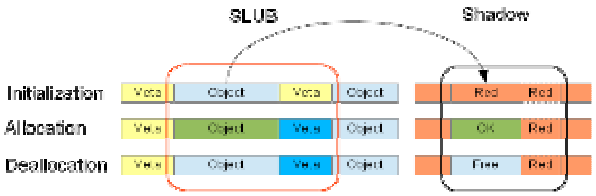

Fig. 8. Sanitizing SLUB allocator.

In recent versions of kernel we also added support for memory pools which are a simple construct on top of Slab. Support for other allocators (e.g. vmalloc) and memory quarantine (to detect use-after-free bugs) is pending.

Apart from allocators, another major difference between userspace and kernel ASan is the bootstrap process. Instrumented code cannot be executed before shadow memory is initialized. In case of userspace, shadow memory setup is handled by runtime library before application start. Obviously we cannot do the same for kernel because the KASan runtime is a part of the kernel itself. Our solution is to have small un-instrumented code to setup the shadow. This code has to be executed before invocation of any instrumented code, therefore it executes at early stage of boot process. Proper initialization of shadow memory cannot be performed at such an early step – instead a special "shadow stub" is used. The stub is represented by a single zero page mapped to entire shadow memory region. After stub is set up, instrumented kernel is able to boot. No errors are detected at this stage as shadow memory is filled with zeros (which effectively means that no memory tracking is done). After linear memory mapping is ready, proper shadow memory can be set up to replace the stub. The sequence of shadow stub and shadow memory region initialization is shown at Fig. 9.

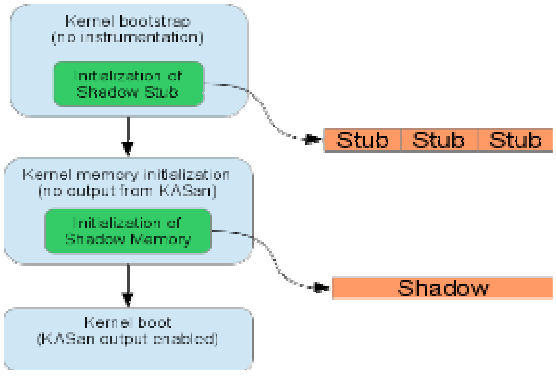

Fig. 9. Usage of "shadow stub" until the shadow memory is correctly initialized.

Compiler is able to instrument only C and C++ code. In opposite to most of userspace applications Linux kernel includes a lot of inline assembly code accessing memory. Thus, invalid accesses initiated by assembly code cannot be detected by KASan. To resolve the problem it is necessary to instrument most frequently used assembly functions manually. The functions are listed at Fig. 10.

atomic_*()
atomic64_*()
test_*_bit()
clear_bit()
xchg()
cmpxchg()
cmpxchg_double()*

Fig. 10. Frequently used inline assembly functions.

ASan supports two types of code instrumentation. Outline instrumentation (historically the first) implies that every memory access is annotated with a function call to check shadow memory. For inline instrumentation, compiler directly inserts checking instructions before memory accesses. This can be much faster (up to 2x on some workloads) but increases code size.

Generally KASan has 3–4 times performance overhead. The chart in Fig. 11 gives the idea of the overhead by comparing performance of 'netperf -l 30' command in normal kernel against instrumented ones. By default, sanitized kernel is built with outline instrumentation but this can be changed during kernel config.

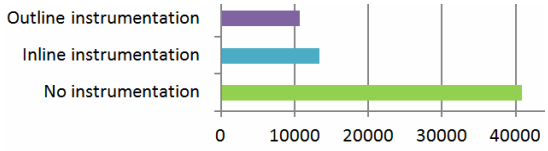

Fig. 11. ASan instrumentation overhead in kernel.

KASan is currently implemented only for x86_64 target but we work on ports to ARM and AArch64.

A number of Linux kernel issues were found with the help of KASan. They include out-of-bounds pointer dereferences in integer ID management library, use-after-free issues in *rmap* and *aio* subsystems, various bugs in generic network layer, out-of-bounds access issues in SCSI, smack and scheduler code.

The most known bug found using KASan is vulnerability in l2tp network layer allowing user privilege escalation (CVE-2014-4943). A fix to the issue can be evaluated as the most significant kernel fix of last seven years.

IV. Conclusion

While AddressSanitizer is a powerful and mature technology its integration to production software systems may not be easy as we discovered on examples of Tizen distribution and Linux kernel. Finally when AddressSanitizer is successfully integrated into QA process it undoubtedly helps to improve software quality by detecting memory corruptions. This article demonstrates problems we faced during ASan integration and provides technical solutions and some ideas for further improvements in this area.

References

[1] D. A. Wheeler, "How to Prevent the next Heartbleed," 29 April 2014. [Online]. Available: http://www.dwheeler.com/essays/heartbleed.html.

[2] K. Serebryany, "AddressSanitizer: A Fast Address Sanity Checker," in USENIX, 2012.

[3] K. Serebryany, "Comparison of Memory Tools," 04 July 2014. [Online]. Available: https://code.google.com/p/address-sanitizer/wiki/ComparisonOfMemoryTools.

[4] H. Boeck, "How Heartbleed could've been found," 7 April 2015. [Online]. Available: https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html.

[5] J. S. Nicholas Nethercote, "How to Shadow Every Byte of Memory Used by a Program," in Proceedings of the 3rd international conference on Virtual execution environments, 2007.

[6] "Tizen on Wikipedia," 18 July 2015. [Online]. Available: https://en.wikipedia.org/wiki/Tizen.

[7] "Zram on Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Zram.

[8] B.P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," in Communications of the ACM 33, 1990.

[9] X. Chen, "MemBrush: A practical tool to detect custom memory allocators in C binaries," in 20th Working Conference on Reverse Engineering, Koblenz, 2013.

[10] M. Larabel, "KernelASan: Bringing Address Sanitizer To The Linux Kernel," 18 July 2014. [Online]. Available: http://www.phoronix.com.