HA^{2}lloc: Hardware-Assisted Secure Allocator

Orlando Arias
University of Central Florida
oarias@knights.ucf.edu

Dean Sullivan
University of Central Florida
dean.sullivan@knights.ucf.edu

Yier Jin
University of Central Florida
yier.jin@eecs.ucf.edu

ABSTRACT

With ever-increasing complexity of software systems, the number of reported security issues increases as well. Among them, memory corruption attacks are a prevalent vector used against today's software stacks. These attacks are repeatedly leveraged to compromise common application software, such as web browsers or document viewers. However, previous work to mitigate memory corruption attacks either suffer from high overhead or can be bypassed by a knowledgeable attacker.

In this work, we introduce HA^{2}lloc, a hardware-assisted allocator that is capable of leveraging an extended memory management unit to detect memory errors in the heap. We also perform some preliminary testing using HA^{2}lloc in a simulation environment and find that the approach is capable of detecting and preventing common memory vulnerabilities.

ACM Reference format:
Orlando Arias, Dean Sullivan, and Yier Jin. 2017. HA^{2}lloc: Hardware-Assisted Secure Allocator. In *Proceedings of HASP '17, Toronto, ON, Canada, June 25, 2017,* 7 pages.
https://doi.org/http://dx.doi.org/10.1145/3092627.3092635

1 INTRODUCTION

As the complexity of modern software increases, the possibility of encountering vulnerabilities that affect platform security increases. These vulnerabilities are estimated to cost the industry billions of dollars every year [1]. For this reason, companies such as Google, Microsoft, and Mozilla have implemented bug bounty programs, where *white hat* hackers are rewarded for finding security issues with their products [2–4]. Likewise, competitions such as Pwn2Own reward *white hat* hackers for their ability to compromise systems. Most of the vulnerabilities reported as part of bug bounty programs and used in competitions like Pwn2Own are memory-related. These vulnerabilities are the result of unsafe usage of languages that allow manual memory management.

Memory errors are prevalent in programs that are written in languages that allow direct access and management of memory. Memory errors can be generalized in two categories: *temporal* and *spatial* [5]. Temporal memory errors occur when the program attempts to utilize an allocation that has already been freed, whereas a spatial error occurs when memory is dereferenced outside valid bounds. At times, memory errors will result in accessing a portion of memory which has not been mapped to the application, resulting in an illegal memory access and a runtime exception being thrown to the application. However, under a sophisticated attacker [6], a memory error can result in security implications for the system such as the possibility to perform code reuse attacks [7] or leak sensitive data [8].

Previous work in academia and industry have used compiler instrumentation or software-based runtime analysis to detect memory errors. However, compiler-based approaches suffer from two issues: the precondition that source code for the application is available, and that the instrumentation is as good as the pointer analysis the compiler performs. Also, software-based runtime analysis introduces large performance penalties and may require a training phase. In this work, we propose a new type of memory allocator which combines both software and hardware elements to provide protection against memory errors while remaining transparent to software running on a platform. We call our memory allocator *HA^{2}lloc*, the *hardware-assisted allocator*. HA^{2}lloc utilizes the facilities of the runtime environment and operating system in combination with an extension to the memory management unit to detect both temporal and spatial memory errors as they occur without the need for compiler instrumentation. We demonstrate the low overhead provided by HA^{2}lloc and how it can be integrated and used to augment other compiler and software-based approaches.

At its heart, HA^{2}lloc employs a modified Memory Management Unit (MMU) in combination with a new memory allocator to detect temporal and spatial memory errors[1]. Our approach utilizes bounds data obtained by the allocator and forwards it to the operating system in order to populate a new set of structures in the MMU. When the MMU handles a memory access that is found in violation with the stored mappings, it triggers a fault which can be handled by the Operating System and the runtime environment.

The main contributions of this paper are:

- The introduction of a new memory protection scheme, HA^{2}lloc, that provides hardware-assisted support to detect memory errors which utilizes metadata obtained from the runtime environment to perform the necessary checks on memory accesses while remaining transparent to the application.
- A study and demonstration of the applicability of the approach as a defense against common attacks, such as virtual function table hijacking, use after free, and counterfeit object oriented programming (COOP).

The rest of this paper is structured as follows. Section 2 provides background information on buffer overflows and their effects. It then introduces previous approaches at protecting systems from these type of vulnerabilities. Section 3 provides a high-level overview of

HASP '17, June 25, 2017, Toronto, ON, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5266-6/17/06…$15.00
https://doi.org/http://dx.doi.org/10.1145/3092627.3092635

[1] At this time, we have only emulated the MMU subsystem as to investigate the feasibility of the approach.

our proposed approach with section 4 describing our implementation. Section 5 provides in-depth testing and evaluation of our platform, including performance metrics and a discussion of its limitations. We then draw conclusions and present future work in Section 6.

2 BACKGROUND

Memory errors continue to be a trend, as the ten years of data collected from the Common Vulnerabilities and Exposures (CVE) database reflect [9]. Figure 1 reflects this data, showing only memory errors with a rating of high to critical. Software exploitation based on stack buffer overflows has dwindled over the years, with use after free vulnerabilities gaining traction and heap buffer overflow vulnerabilities maintaining steady momentum. We notice that some of the most powerful attacks are heap based, as we see an increasing trend in spatial and temporal heap-based vulnerabilities.

2.1 Example Vulnerability

Consider the sample code shown in Listing 1. Here, we demonstrate both temporal and spatial memory errors. There is a potential use after free vulnerability, as any of the objects stored in the `c` array may actually get deallocated before their member functions are called, resulting in the temporal memory error. There is also a potential spatial memory error by calling the `load_buffer()` function with a parameter that is larger in size than the buffer contained in the object. This results in a heap buffer overflow.

Listing 1: A small, vulnerable interpreter

```
#include <cstring>

class base {
  public:
    virtual void function() { ; }
    virtual void load_buffer(const char* buffer)
        = 0;
};

class derived : public base {
  char buffer[128];
  public:
    void function() { buffer[0] = '\0'; }
    void load_buffer(const char* buffer) {
      strcpy(this->buffer, buffer);
    }
};

int main(int argc, char* argv[]) {
  base* c[] = {nullptr, nullptr};
  char* p = argv[2];
  char m;

  while(*p) {
    switch(m = *p++) {
      case 'n':
      case 'N':
        if(!c[m == 'N'])
          c[m == 'N'] = new derived;
        break;
      case 'l':
      case 'L':
        c[m == 'L']->load_buffer(argv[1]);
        break;
      case 'f':
      case 'F':
        c[m == 'F']->function();
        break;
      case 'd':
      case 'D':
        delete c[m == 'D'];
        break;
    }
  }
  return 0;
}
```

An attacker can then utilize these vulnerabilities in order to corrupt memory in the heap. If allocation headers are kept near the allocations, then the buffer overflow vulnerability can be leveraged to inject a corrupted header. Furthermore, by careful manipulation of the allocations in the heap, a new `vtable` pointer can be injected to gain arbitrary control flow through a COOP-style attack [10].

Spatial memory errors can also result in the disclosure of sensitive information such as the base address of critical data structures or code pointers, thereby allowing the attacker to bypass randomization schemes that attempt to hide the locations of code and data segments such as ASLR [11]. As seen in the example, spatial memory errors can be exploited to overwrite these critical data structures or code pointers, allowing for information flow attacks or control flow attacks. An attacker is able to utilize temporal memory errors as a way to redirect control flow by injecting control flow data, such as a `vtable` pointer, into the reallocated memory region the stale object used to occupy.

2.2 Previous Work

Baggy Bounds Checking [12] introduces bounds checking for arrays in a granular fashion. Instead of keeping exact bounds for each array, it pads the allocation into bounds that are powers of two. This is done to reduce the overhead of the metadata by storing the exponent of the allocation only. On a 32 bit system, only 5 bits are needed to save the data and at storage time, one full byte is used. C library functions that deal with arrays, such as `strcpy()` and `memcpy()`, are provided with wrappers that check the bounds of the arrays before executing them. However, the mechanism is unable to prevent access errors when the buffer is located within an object such as a struct. Baggy Bounds Checking is a compiler based solution and thus requires binaries to be instrumented at compile time: source code is required. Unfortunately, no tools have been released to the general market. Looseness on the stored metadata also results in some checks being inaccurate. Performance wise, a 60% overhead is reported on a modified SPEC2000 suite and a 15% overhead in some Olden benchmarks.

AddressSanitizer [13] provides a method to instrument bounds check for software written in C and C++. It is implemented as a compiler pass and a runtime library. A portion of memory is dedicated as shadow memory, where metadata about arrays are stored. The memory is mapped into intervals of N bytes, and the mapping into the shadow area computed as $Addr >> Scale + Offset$ where Scale is given by N. If the transformation is applied to the shadow memory area, the resulting address will point to a portion of memory which is not mapped into the process's virtual address space, thus generating an access violation. AddressSanitizer provides a runtime library to aid with dynamic allocations, providing new versions of the `malloc()` family of functions and `free()`. The new

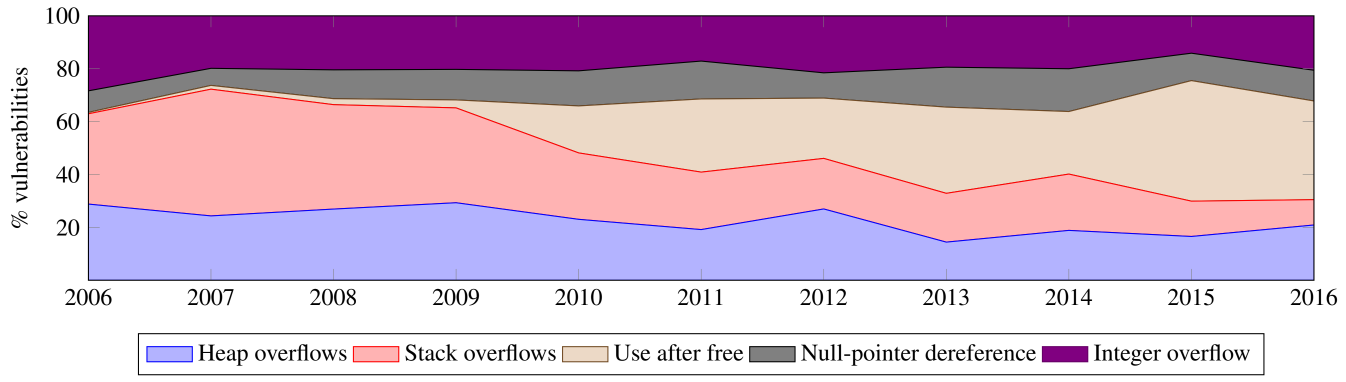

Figure 1: Trends in memory errors collected from the CVE database [9]. We show trends in memory errors in the last ten years that have resulted in a software vulnerability. Observable is how stack exploits have dwindled in favor of heap-based exploits.

allocator functions provide redzones around the returned region. These redzones are flagged as unaddressable and are used to store data from the allocator. The new implementation of `free()` poisons these redzones and puts them into a quarantine mode. Redzones are also added to buffers stored in stack frames. AddressSanitizer, however, presents a few false negatives and false positives, such as unaligned accesses that are partially out of bounds, accesses that fall too far away from the object bounds that may land in a different valid location, and load widening.

Sarbinowski et al propose VTPin in [14] as a way to counter some use after free exploits that result from temporal memory errors. VTPin provides a small library that intercepts calls to the allocator. Specifically, when a deallocation takes place, the VTPin takes control of the allocation and infers whether the deallocation corresponds to a C++ object. If it is, VTPin performs an in-place reallocation, leaving sufficient area to store a new set of virtual function table pointers. These point to an implementation controlled virtual function table. The in-place reallocation ensures that the virtual table pointer area is never reutilized, thus an attacker is unable to overwrite the virtual table pointer area by means of conventional heap spraying attacks such as Heap Feng Shui [6].

Watchdog [15] and WatchdogLite [16] propose a mechanism to store and check bounds data of a pointer or array with some hardware acceleration by using the SIMD extensions of x86 and x86_64 processors. This provides protection against spatial memory errors. Intel MPX [17] provides functionality similar to that of WatchdogLite, with the distinction that a dedicated set of registers, instructions and hardware exceptions were added to the processor. Intel MPX is available on 6th generation and newer processors. Being ISA-based, these approaches require compiler instrumentation for them to be of use.

Woodruff et al introduce Capability Hardware Enhanced RISC Instructions (CHERI) as a method to add *capabilities* to memory accesses in [18]. *Capabilities* are defined as the right to perform an action or set of actions to a given object. Furthermore, capabilities can be transferred between objects. In CHERI's case, the capabilities define the right of an instruction to make a memory access. For the purposes of implementation, CHERI is built as a coprocessor in a MIPS64 compatible core. Much like the previously mentioned approaches, compiler support is necessary to issue the necessary coprocessor instructions in a program. For this purpose, the authors utilize the LLVM compiler infrastructure in order to instrument source code.

2.3 Limitations of Previous Work

Compiler-based approaches such as AddressSanitizer [13], Baggy Bounds Checking [12], Watchdog [15], WatchdogLite [16], and Intel's MPX [17] inherently suffer from the outset as source code is required in order to instrument applications. Furthermore, the instrumentation is only as good as the correctness and completeness of the pointer analysis the compiler can perform. Unfortunately, pointer analysis has proven to be undecidable for the general case [19], and different algorithms suffer from either runtime or spatial considerations [20]. As such, compilers will perform a safe overestimation which can lead to incorrect instrumentation.

Herein lies the main issue with current compiler-based metadata approaches. Because we can not determine whether two symbols alias to the same value, we are unable to properly propagate metadata on this symbol for the general case. As such, there are instances where the information needed to perform the check is not available or inaccurate. Since compilers err on the side of safety, any performed check with incomplete or inaccurate metadata will pass, allowing temporal and spatial memory errors to occur.

Other approaches attempt to address either spatial or temporal memory errors. For example, although VTPin [14] ensures that the portions of an object that correspond to a virtual function table pointer can not be overwritten by subsequent allocations, it is unable to protect these areas against corruption that happens due to conventional heap buffer overflows. We were able to demonstrate this by crafting our own implementation of VTPin and constructing a vulnerable program that allocates two objects in the heap. We then free one of the objects and utilize a heap buffer overflow vulnerability in the other object to write into the reallocation made by VTPin. This results in the virtual function table pointer kept by VTPin being corrupted, resulting in arbitrary code execution from an attacker's perspective. We should note that this attack is still possible even if the object is not deallocated.

3 PROPOSED APPROACH

Although compiled languages such as C and C++ often lose information on arrays when the final binary is built, such information may be reconstructed at runtime. For example, when a program dynamically allocates memory, the allocator has knowledge of both the allocation size and the address at which the allocation was made. We leverage this runtime information to gather the necessary metadata to enforce our buffer overflow protection and our temporal memory safety scheme.

3.1 Dynamic Memory Allocations

Modern computing systems implement process isolation by providing each process with its own virtual address space. In an `AMD64`-based system, each process is given a potential 48bit address space. However, no application is given a full address space when executing, as systems do not contain enough physical memory to support this. As such, applications are given the ability to dynamically request memory from the system. Enter the `malloc()` family of functions from the C library, and the `new` operator from the C++ language. With this, an application is able to expand its memory footprint by adding memory to the *heap*.

An allocator manages the heap memory for a process. The allocator is provided by the runtime environment, namely the C library in combination with the operating system, and it is completely transparent to the program. When a process deallocates memory using the `free()` function or the `delete` keyword, the allocator flags that portion of memory as unused, and can potentially cache it for future allocations. If there is not enough unused memory in the heap to satisfy a request, then the allocator proceeds to request more memory from the operating system utilizing the *system call interface*.

Internally, an allocator utilizes a series of data structures to keep a record of which allocations made by the application are currently active and which ones are freed. This data structure is called an *allocation header*. The way the allocator manages the allocation headers and the information they contain are specific to the allocator implementation itself. For example, some allocators, such as dlmalloc and derivatives [21], choose to keep allocation headers in front of the allocated space. This has the benefit of the allocator quickly being able to access information about the allocation by offsetting from a pointer to the allocated space. Unfortunately, a heap buffer overflow can easily corrupt adjacent allocation headers. Other allocators, such as OpenBSD's allocator, keep the allocation headers in a separate portion of memory [22]. This portion of memory is randomly mapped to the application and kept in a different memory area from the allocation itself. Although this secures allocation headers from being corrupted, the mechanism requires a search to be performed looking for the allocation header that matches the allocation itself. However, there are still common elements found in allocation headers. The size of of every allocation the application makes, the area of memory occupied by the allocation, and whether the allocated area has been freed or not is kept.

3.2 Design Constraints

With HA^{2}lloc, we wish to provide a drop-in mechanism that is compatible with existing applications without needing to rewrite or recompile them. For this purpose, we constrain our design to meet the following points:

- Transparency: The system must be completely transparent to applications. An application which exhibits legal behavior must not be affected in operation by the buffer overflow protection mechanism, nor should the application be able to infer it is running under the mechanism.
- Portability: Existing applications must work under the system without any type of modification to their source code and/or binaries. Applications are not to be modified at load time either.
- Integration: The mechanism must be easily integrated in an existing operating system and runtime environment with minor modifications. As long as the underlying hardware platform supports the mechanism, it should work without triggering any false-positives.

Given these constraints, compiler modifications are not allowed, as these will reflect a change in the binaries that get deployed on the system, violating the *Portability* requirement. Only modifications to the runtime, the operating system and underlying hardware platform are allowed. As such, we assume that an application will utilize the resources provided by the runtime environment and operating system, and conform to standard architectural and ABI conventions with special function registers.

In order to design HA^{2}lloc we observe the following:

(1) The internal data structures in the allocator have knowledge of the place where the allocated memory resides at and their sizes.
(2) The allocator must communicate with the operating system to request more memory when needed.

We utilize these observations in the next subsection to introduce the concepts behind HA^{2}lloc.

3.3 Introduction to HA^{2}lloc

We show a high level overview of HA^{2}lloc in Figure 2. At its heart, HA^{2}lloc provides a security aware allocator which separates allocation headers from the actual allocations in the heap. In doing so, we obtain two benefits. First, allocation headers can not be corrupted by conventional heap overflows. This aids with the integrity of the allocator. Secondly, it allows us to flag pages that have been specifically added to a process for the purposes of dynamic allocations.

When an application requests memory from the system using the `malloc()` family of functions, HA^{2}lloc's allocator handles the request. Besides performing a request to the operating system to allocate new pages to the application, HA^{2}lloc also forwards allocation metadata to the operating system itself. The allocation metadata consists of the size of the allocation and a possible base address in a page. The operating system records the allocation metadata on the page table entries used by the memory management unit as well as flag the associated pages as heap pages.

Furthermore, we randomize the base address of allocations within the process's virtual address space. In doing so, we introduce an extra

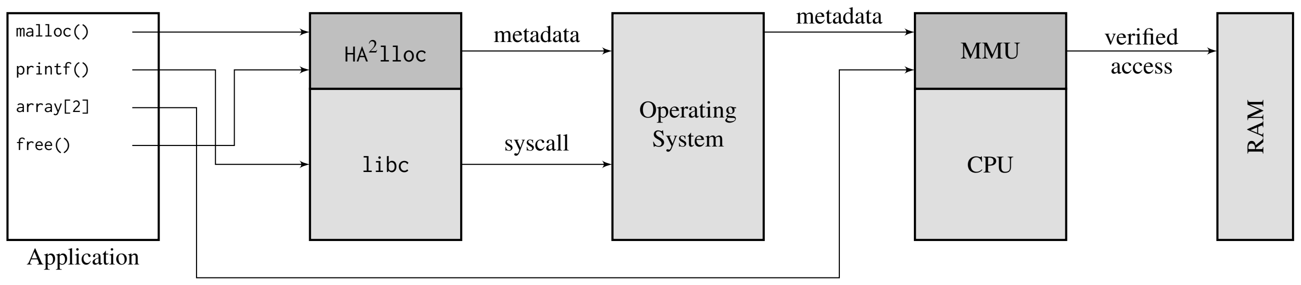

Figure 2: Overview of HA^{2}lloc: HA^{2}lloc provides the facilities required by an application to perform dynamic memory allocations whilst forwarding allocation metadata to the operating system. The operating system itself stores this information in the page table for the application. An extended MMU is then capable of utilizing the information to check memory accesses performed by the application.

layer of unpredictability to the allocator. That is, for two independent runs of the same program, two very different heap address maps are generated. This further allows us to mitigate heap-spraying style attacks, such as heap feng shui [6].

Since HA^{2}lloc only provides the means to perform allocations, application software can go on to utilize other facilities provided by the system libraries. The system libraries can utilize HA^{2}lloc's facilities to perform any dynamic allocations.

Any access the application performs to heap-mapped pages can then be verified by the MMU. The validation step remains transparent to the application, as it is performed directly by the MMU subsystem. Since the page table contains bounds information, the MMU can utilize this information to check accesses to heap mapped pages. If the access occurs within the recorded bounds, it is allowed. Otherwise, a fault is triggered and a signal is sent to the operating system.

We also need to be able to handle temporal memory errors. In order to do so, we must be able to handle any deallocations made by an application. When the process relinquishes an allocation by either calling the `free()` function, the `realloc()` function, or the `delete` keyword in C++, HA^{2}lloc signals the operating system, forwarding information on the ongoing deallocation. The operating system in turn eliminates the allocation entry from the page table. If no more allocations reside in that particular table, the operating system unmaps the page from the process. The unmapped virtual address space is never reused.

When the process attempts a memory access to a deallocated area in the heap, one of two things will happen: either the page is unmapped triggering an illegal memory access, or the MMU is unable to find the bounds of the accessed address in the page table, triggering a similar fault. The operating system then is able to handle the fault accordingly, by either terminating the application or throwing a signal to the application.

4 IMPLEMENTATION DETAILS

4.1 The HA^{2}lloc Allocator

Linux-based systems that utilize the GNU C Library use a modified dlmalloc as the base to manage heap allocations [21]. Allocators based on dlmalloc have the characteristic that they keep allocation metadata in front of the allocation that is returned to callers. The allocation metadata, or allocation header contains information on the size of the allocation, the next allocation bucket, and some other flags. Having the allocation header in front of the allocation allows the allocator functions to quickly obtain data from an allocation.

Although simple in design and fast in execution, a well versed attacker is able to exploit this allocator behavior to spray the heap and fool the allocator into thinking regions are allocated when they are not. Furthermore, heap buffer overflows allow an attacker to corrupt allocation headers, further enhancing their control over the application.

4.1.1 Allocating Memory. For this purpose, HA^{2}lloc's allocator keeps the allocation metadata separate from the allocations themselves. Upon initialization, HA^{2}lloc's allocator maps a page of memory where it keeps all allocation headers. Whenever a program requests memory through the use of `malloc()`, `calloc()`, or `realloc()`, HA^{2}lloc requests memory from the operating system and creates a new allocation header. The allocator header is stored as part of a hash table. In order to handle collisions in the hash table, we utilize a red-black tree [23] on each bucket. This allows us to perform operations on the data structure in $\mathcal{O}\log n$ computational time in contrast to the amortized $\mathcal{O}n$ computational time that would result in handling collisions and resizing the hash table. Furthermore, by performing operations in this fashion in the hash table we can reduce the number of semaphores used in the allocation data structures, allowing for better parallelism in multi-threaded applications. Once the allocation is made and the header is constructed HA^{2}lloc returns a pointer to the allocation to the user.

Of importance to HA^{2}lloc is how pages are mapped to the application. Ideally, we would like to randomize the addresses of the pages mapped to the application whilst still ensuring that large allocations remain continuous in memory. Preliminary testing shows that Linux's `sys_mmap` does not attempt to randomize the addresses of the pages returned. The first mapped page has a relative random address. However, subsequent calls to `mmap()` will return pages at a fixed offset from the first page. This is detrimental to the security of our allocator, as all allocations would be in a predictable memory address. For this purpose, we introduce a new system call in the Linux kernel which performs a function similar to that of `sys_mmap` but it returns pages in a randomized fashion. We forward information about the desired allocation size to the kernel using this mechanism.

This information is used by the HA^{2}lloc's hardware subsystem to transparently perform bounds check in heap accesses (see Section 4.2).

4.1.2 Deallocating Memory. When an application deallocates memory, HA^{2}lloc removes the allocation header from the hash table, modifying the red-black tree if necessary. The removed allocation headers are added to a linked list to be reused by new allocations. The pages corresponding to these allocations are unmapped from the program. We ensure that these pages are never mapped to the program again by keeping a list of pages unmapped by the application within the virtual address map kept by the kernel in the process control block. In doing so, temporal memory errors result in an illegal memory access, triggering a segmentation fault.

4.2 HA^{2}lloc's Hardware Subsystem

HA^{2}lloc's Hardware Subsystem has yet to be implemented and tested. At its base, we extend the MMU to add one extra bit to flag heap allocated pages. Since heap pages are allocated using a new system call, no extra overhead is incurred in this flagging mechanism. We keep bounds information at the page level by associating a 32bit word to a heap page table entry. We illustrate the encoding in Figure 3.

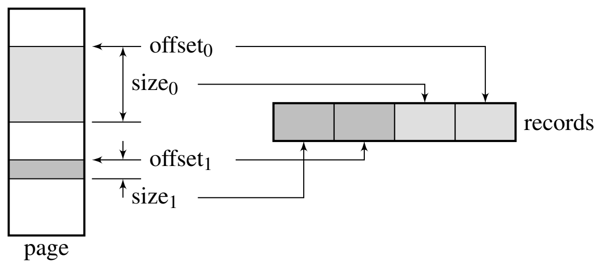

Figure 3: Bounds encoding mechanism used in HA^{2}lloc. A 32bit **word is associated with every heap page table entry. This word contains bounds information used by the MMU to perform checks on heap accesses.**

In order to record information on buffer sizes and offsets into the pages, we first analyze a few architectural constraints an allocator must follow. Type information is generally lost when compiling C code. Furthermore, the `malloc()` family of functions do not receive type information regarding the allocation that is being made. As such, these functions must assume a worse case scenario alignment for the datatype that is being allocated, both in terms of performance and ISA limitations. In C++, the `new` keyword could potentially use type information and specialize the allocation to better suit the datatype, but to the best of our knowledge, no C++ allocator performs this optimization.

For HA^{2}lloc, we assume a worst case alignment of 16B, given that this is the alignment required for common instruction set extensions such as Intel AVX [24]. As such, in a 4096B page, we can start at 256 different offsets. Consequently, we divide the 32bit word into 8bit subsections. We then group the subsections in pairs, with the lower byte denoting in which 16B block the allocation into the page starts, or the *offset* into the page, and the upper byte the number of 16B blocks covered by the allocation, or the *size* of the allocation. This means that we can potentially have up to two allocations per page. For two small allocations, we are then able to leave unused space between them, which can serve as a *red zone* to catch overruns. Multi-page allocations are handled in a similar fashion. Since the size field can cover the entire page, we can let the size field encompass the entire page, indicating that it covers a buffer.

When a memory access occurs to a heap-flagged page, the MMU utilizes the offset and size information recorded on the associated word to the page table entry and checks whether the access is within bounds specified for the allocations in the page. If it is, then virtual to physical address translation occurs as normal and the memory access is allowed. On the other hand, if the check fails, it is deemed to be caused by an illegal access. The MMU triggers a fault at this point, which must be handled by the operating system.

5 PRELIMINARY EVALUATION

A preliminary evaluation of our prototype implementation shows that for large allocations, HA^{2}lloc is faster than the `dlmalloc` implementation used in `glibc`. This is because `glibc` will scan through a circular list of freed allocations before mapping new heap pages to the application. For smaller allocations, `glibc` will expand the heap using the `sbrk` system call and perform the smaller allocations in that area. Since `glibc` can expand the heap multiple pages at a time using the `sbrk` system call, it can cache pages to be used by subsequent allocations and avoid expensive context switches.

Method	Temporal	Spatial
Baggy Bounds Checking [12]	no	yes†
AddressSanitizer [13]	no	yes†
VTPin [14]	yes	no
Watchdog [15]	no	yes†
WatchdogLite [16]	no	yes†
Intel MPX [17]	no	yes†
CHERI [18]	no	yes†
Our approach	yes	yes‡

† Requires instrumentation.

‡ In our current prototyping phase, bounds check is performed in a simulated environment and not implemented in a hardware MMU.

Table 1: Comparison between approaches

Table 1 offers a comparison between our protection mechanism and previous work. When running our sample vulnerable application on Section 2 we found HA^{2}lloc to be capable of detecting and preventing both the temporal and spatial memory errors. We also found that the vulnerabilities in the program were readily exploitable when testing against `glibc`'s `dlmalloc`. We also found that our reimplementation of VTPin was able to prevent the temporal memory error as long as the spatial memory error vulnerability was not triggered.

6 CONCLUSIONS AND FUTURE WORK

In this work, we present HA^{2}lloc, a secure memory allocator that utilizes an extended memory management unit to detect both temporal and spatial memory errors in the heap. We present the concepts behind HA^{2}lloc as well as preliminary testing of its implementation. We also compare HA^{2}lloc to previously proposed mechanisms in terms of coverage and deployability.

Future work for HA^{2}lloc includes the implementation of the memory management unit subsystem in order to test the effectiveness of the spatial memory error detection as well as any incurred overhead from these checks. Furthermore, we wish to be able to test reported vulnerabilities against HA^{2}lloc to further validate its usefulness. Lastly, we plan to extend HA^{2}lloc to also include stack-based buffers, as to provide a complete temporal and spatial memory error detection solution.

7 ACKNOWLEDGEMENTS

This paper is partially supported by the Department of Energy through the Early Career Award (DE-SC0016180). Mr. Orlando Arias is also supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 1144246. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the Department of Energy.

REFERENCES

[1] R. Telang and S. Wattal, "An empirical analysis of the impact of software vulnerability announcements on firm stock price," *IEEE Transactions on Software Engineering*, vol. 33, no. 8, pp. 544–557, 2007.
[2] Mozilla Foundation, "Mozilla security bug bounty program," https://www.mozilla.org/en-US/security/bug-bounty/.
[3] Microsoft Corporation, "Microsoft bounty programs," https://technet.microsoft.com/en-us/library/dn425036.aspx.
[4] Google, Inc., "Google application security," https://www.google.com/about/appsecurity/.
[5] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 48–62.
[6] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, 2007.
[7] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.
[8] P. Ducklin, "Anatomy of a data leakage bug – the OpenSSL "heartbleed" buffer oveflow," 2014, https://nakedsecurity.sophos.com/2014/04/08/anatomy-of-a-data-leak-bug-openssl-heartbleed/.
[9] The MITRE Corporation, "Common vulnerabilities and exposures," https://cve.mitre.org/.
[10] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 745–762.
[11] P. Team, "Pax address space layout randomization (aslr)," 2003.
[12] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*, 2009, pp. 51–66.
[13] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.
[14] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos, "Vtpin: Practical vtable hijacking protection for binaries," in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: ACM, 2016, pp. 448–459. [Online]. Available: http://doi.acm.org/10.1145/2991079.2991121
[15] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 189–200.
[16] ——, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 175:175–175:184. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544147
[17] P. Guide, "Intel® 64 and ia-32 architectures software developerâĂŹs manual," *Volume 3B: System programming Guide, Part*, vol. 2, 2011.
[18] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665740
[19] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.
[20] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '01. New York, NY, USA: ACM, 2001, pp. 54–61. [Online]. Available: http://doi.acm.org/10.1145/379605.379665
[21] D. Lea and W. Gloger, "glibc `malloc()`," http://malloc.de/en/.
[22] P.-H. Kamp, D. Miller, M. Dempsky, and O. Moerbeek, "Openbsd `malloc()`," http://bxr.su/OpenBSD/lib/libc/stdlib/malloc.c.
[23] R. Sedgewick and L. J. Guibas, "A dichromatic framework for balanced trees," *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, vol. 00, pp. 8–21, 1978.
[24] Intel Corporation, "Intel Architecture Instruction Set Extensions Programming Reference," December 2016, document Number: 319433-028.

LAZARUS: Practical Side-channel Resilient Kernel-Space Randomization

David Gens[1], Orlando Arias[2], Dean Sullivan[2], Christopher Liebchen[1], Yier Jin[2], and Ahmad-Reza Sadeghi[1]

[1] CYSEC/Technische Universität Darmstadt, Germany.
{david.gens,christopher.liebchen,ahmad.sadeghi}@trust.tu-darmstadt.de
[2] University of Central Florida, Orlando, FL, USA.
{oarias,dean.sullivan}@knights.ucf.edu,yier.jin@eecs.ucf.edu

Abstract. Kernel exploits are commonly used for privilege escalation to take full control over a system, e.g., by means of code-reuse attacks. For this reason modern kernels are hardened with kernel Address Space Layout Randomization (KASLR), which randomizes the start address of the kernel code section at boot time. Hence, the attacker first has to bypass the randomization, to conduct the attack using an adjusted payload in a second step. Recently, researchers demonstrated that attackers can exploit unprivileged instructions to collect timing information through side channels in the paging subsystem of the processor. This can be exploited to reveal the randomization secret, even in the absence of any information-disclosure vulnerabilities in the software.
In this paper we present *LAZARUS*, a novel technique to harden KASLR against paging-based side-channel attacks. In particular, our scheme allows for fine-grained protection of the virtual memory mappings that implement the randomization. We demonstrate the effectiveness of our approach by hardening a recent Linux kernel with LAZARUS, mitigating all of the previously presented side-channel attacks on KASLR. Our extensive evaluation shows that LAZARUS incurs only 0.943% overhead for standard benchmarks, and therefore, is highly practical.

Keywords: KASLR, Code-Reuse Attacks, Randomization, Side Channels

1 Introduction

For more than three decades memory-corruption vulnerabilities have challenged computer security. This class of vulnerabilities enables the attacker to overwrite memory in a way that was not intended by the developer, resulting in a malicious control or data flow. In the recent past, kernel vulnerabilities became more prevalent in exploits due to advances in hardening user-mode applications. For example, browsers and other popular targets are isolated by executing them in a sandboxed environment. Consequently, the attacker needs to execute a privilege-escalation attack in addition to the initial exploit to take full control over the system [4, 17, 18, 19]. Operating system kernels are a natural target for attackers because the kernel is comprised of a large and complex code base, and exposes a rich set of functionality, even to low privileged processes. Molinyawe et al. [20] summarized the techniques used in the Pwn2Own exploiting contest, and concluded that a kernel exploit is required for most privilege-escalation attacks.

In the past, kernels were hardened using different mitigation techniques to minimize the risk of memory-corruption vulnerabilities. For instance, enforcing the address space to be writable or executable (W⊕X), but never both, prevents the attacker from injecting new code. Additionally, enabling new CPU features like Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Protection (SMEP) prevents certain classes of user-mode-aided attacks. To mitigate code-reuse attacks, modern kernels are further fortified with kernel Address Space Layout Randomization (KASLR) [2]. KASLR randomizes the base address of the code section of the kernel at boot time, which forces attackers to customize their exploit for each targeted kernel. Specifically, the attack needs to disclose the randomization secret first, before launching a code-reuse attack.

In general, there are two ways to bypass randomization: (1) brute-force attacks, and (2) information-disclosure attacks. While KASLR aims to make brute-force attacks infeasible, attackers can still leverage information-disclosure attacks, e.g., to leak the randomization secret. The attacker can achieve this by exploiting a memory-corruption vulnerability, or through side channels. Recent research demonstrated that side-channel attacks are more powerful, since they do not require any kernel vulnerabilities [6, 8, 10, 13, 23]. These attacks exploit properties of the underlying micro architecture to infer the randomization secret of KASLR. In particular, modern processors share resources such as caches between user mode and kernel mode, and hence, leak timing information between privileged and unprivileged execution. The general idea of these attacks is to probe different kernel addresses and measure the execution time of the probe. Since the timing signature for valid and invalid kernel addresses is different, the attacker can compute the randomization secret by comparing the extracted signal against a reference signal.

The majority of side-channel attacks against KASLR is based on *paging* [8, 10, 13, 23]. Here, the attacker exploits the timing difference between an aborted memory access to an unmapped kernel address and an aborted memory access to a mapped kernel address. As we eloberate in the related work Section 7 the focus of the existing work is on attacks, and only include theoretical discussions on possible defenses. For instance, Gruss et al. [8] briefly discuss an idea similar to our implemented defense by suggesting to completely un-map the kernel address space when executing the user mode as it is done in iOS on ARM [16]. However, as stated by the authors [8] they did not implement or evaluate the security of their approach but only provided a simulation of this technique to provide a rough estimation of the expected run-time overhead which is around 5% for system call intensive applications.

Goal and Contributions The goal of this paper is to prevent kernel-space randomization approaches from leaking side-channel information through the paging subsystem of the processor. To this end, we propose *LAZARUS*, as a novel real-world defense against paging-based side-channel attacks on KASLR. Our software-only defense is based on the observation that all of the presented attacks have a common source of leakage: information about randomized kernel addresses is stored in the paging caches of the processor while execution continues in user mode. More specifically, the processor keeps paging entries for recently used addresses in the cache, regardless of their associated privilege level. This results in a timing side channel, because accesses for cached entries are faster than cache misses. Our defense separates paging entries according to their privilege level in caches, and provides a mechanism for the kernel to achieve this efficiently in software. LAZARUS only separates those parts of the address space which might reveal the randomization secret while leaving entries for non-randomized memory shared. Our benchmarks show that this significantly reduces the performance overhead. We provide a prototype implementation of our side-channel defense, and conduct an extensive evaluation of the security and performance of our prototype for a recent kernel under the popular Debian Linux and Arch Linux distributions.

To summarize, our contributions are as follows:

- **Novel side-channel defense.** We present the design of *LAZARUS*, a software-only protection scheme to thwart side-channel attacks against KASLR based on paging.
- **Protoype Implementation.** We provide a fully working and practical prototype implementation of our defense for a recent Linux kernel version 4.8.
- **Extensive Evaluation.** We extensively evaluate our prototype against all previously presented side-channel attacks and demonstrate that the randomization secret can no longer be disclosed. We re-implemented all previously proposed attacks on KASLR for the Linux kernel. We additionally present an extensive performance evaluation and demonstrate high practicality with an average overhead of only 0.943% for common benchmarks.

2 Background

In this section, we first explain the details of modern processor architectures necessary to understand the remainder of this paper. We then explain the different attacks on KASLR presented by related work.

2.1 Virtual Memory

Virtual memory is a key building block to separate privileged system memory from unprivileged user memory, and to isolate processes from each other. Virtual memory is implemented by enforcing an indirection between the address space of the processor and the physical memory, i.e., every memory access initiated by the processor is mediated by a piece of hardware called the Memory Management Unit (MMU). The MMU translates the virtual address to a physical address, and

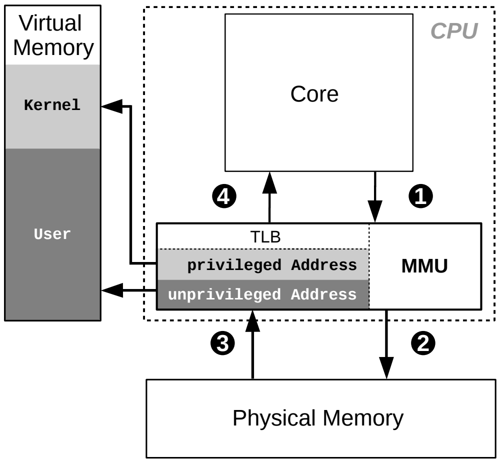

Fig. 1. When virtual memory is active, all memory accesses of the processor are mediated by the MMU ❶: it loads the associated page-table entry ❷ into the TLB from memory, checks the required privilege level ❸, and translates the virtual memory address into the corresponding physical memory address if and only if the current privilege level of the processor matches the required privilege level ❹.

enforces access control based on permissions defined for the requested address. The translation information as well as the access permissions are stored in a hierarchical data structure, which is maintained by the kernel, called the page table. The kernel isolates processes from each other by maintaining separate page tables for each process, and hence, different permissions. In contrast to processes, the kernel is not isolated using a separate page table but by setting the supervisor bit in page-table entries that translate kernel memory. In fact, each process page table contains entries that map the kernel (typically in the top part of the virtual address space). This increases the performance of context switches between the kernel and user applications because replacing the active page table forces the MMU to evict entries from its internal cache, called Translation Lookaside Buffer (TLB). The TLB caches the most recent or prominent page table entries, which is a sensible strategy since software usually exhibits (spatial or temporal) *locality*. Hence, all subsequent virtual-memory accesses, which are translated using a cached page-table entry, will be handled much faster.

Figure 1 shows the major components of virtual memory and their interaction. In the following we describe the MMU and the TLB in detail and explain their role in paging-based side-channel attacks.

The Central Processing Unit (CPU) contains one or more execution units (cores), which decode, schedule, and eventually execute individual machine instructions, also called operations. If an operation requires a memory access, e.g., load and store operations, and the virtual memory subsystem of the processor is enabled, this access is mediated by the MMU (Step ❶). If the page-table entry for the requested virtual address is not cached in the TLB, the MMU loads the entry into the TLB by traversing the page tables (often called a *page walk*) which reside in physical memory (Step ❷). The MMU then loads the respective page-table entry into the TLBs (Step ❸). It then uses the TLB entries to look up the physical address and the required privilege level associated with a virtual address (Step ❹).

2.2 Paging-based Side-channel Attacks on KASLR

All modern operating systems leverage kernel-space randomization by means of kernel code randomization (KASLR) [2, 11, 14]. However, kernel-space randomization has been shown to be vulnerable to a variety of side-channel attacks. These attacks leverage micro-architectural implementation details of the underlying hardware. More specifically, modern processors share virtual memory resources between privileged and unprivileged execution modes through caches, which was shown to be exploitable by an user space adversary.

In the following we briefly describe recent paging-based side-channel attacks that aim to disclose the KASLR randomization secret. All these attacks exploit the fact that the TLB is shared between user applications and the kernel (cf., Figure 1). As a consequence, the TLB will contain page-table entries of the kernel after switching the execution from kernel to a user mode application. Henceforth, the attacker uses special instructions (depending on the concrete side-channel attack implementation) to access kernel addresses. Since the attacker executes the attack with user privileges, the access will be aborted. However, the time difference between access attempt and abort depends on whether the guessed address is cached in the TLB or not. Further, the attacker can also measure the difference in timing between existing (requiring a page walk) and non-existing mappings (immediate abort). The resulting timing differences can be exploited by the attacker as a side channel to disclose the randomization secret as shown recently [8, 10, 13, 23].

Page Fault Handler (PFH) Hund, et al. [10] published the first side-channel attack to defeat KASLR. They trigger a page fault in the kernel from a user process by accessing an address in kernel space. Although this unprivileged access is correctly denied by the page fault handler, the TLBs are queried during processing of the memory request. They show that the timing difference between exceptions for unmapped and mapped pages can be exploited to disclose the random offset.

Prefetch Instruction Furthermore, even individual instructions may leak timing information and can be exploited [8]. More specifically, the execution of the `prefetch` instruction of recent Intel processors exhibits a timing difference, which depends directly on the state of the TLBs. As in the case of the other side-channel attacks, this is used to access privileged addresses by the attacker.

Since this access originates from an unprivileged instruction it will fail, and according to the documentation the processor will not raise an exception. Hence, its execution time differs for cached kernel addresses. This yields another side channel that leaks the randomization secret.

Intel's TSX Transactional memory extensions introduced by Intel encapsulate a series of memory accesses to provide enhanced safety guarantees, such as rollbacks. While potentially interesting for the implementation of concurrent software without the need for lock-based synchronization, erroneous accesses within a transaction are not reported to the operating system. More specifically, if the MMU detects an access violation, the exception is masked and the transaction is rolled back silently. However, an adversary can measure the timing difference between two failing transactions to identify privileged addresses, which are cached in the TLBs. This enables the attacker to significantly improve over the original page fault timing side-channel attack [13, 23]. The reason is that the page fault handler of the OS is never invoked, significantly reducing the noise in the timing signal.

3 LAZARUS

In this section, we give an overview of the idea and architecture of LAZARUS, elaborate on the main challenges, and explain in detail how we tackle these challenges.

3.1 Adversary Model and Assumptions

We derive our adversary model from the related offensive work [6, 8, 10, 13, 23].

– **Writable $\oplus$ Executable Memory**. The kernel enforces Writable $\oplus$ Executable Memory (W$\oplus$X) which prevents code-injection attacks in the kernel space. Further, the kernel utilizes modern CPU features like SMAP and SMEP [12] to prevent user-mode aided code-injection and code-reuse attacks.
– **Kernel Address Space Layout Randomization (KASLR)**. The base address of the kernel is randomized at boot time [2, 14].
– **Absence of Software-based Information-disclosure Vulnerability**. The kernel does not contain any vulnerabilities that can be exploited to disclose the randomization secret.
– **Malicious Kernel Extension**. The attacker cannot load malicious kernel extensions to gain control over the kernel, i.e., only trusted (or signed) extensions can be loaded.
– **Memory-corruption Vulnerability**. This is a standard assumption for many real-world kernel exploits. The kernel, or a kernel extension contains a memory-corruption vulnerability. The attacker has full control over a user-mode process from which it can exploit this vulnerability. The vulnerability

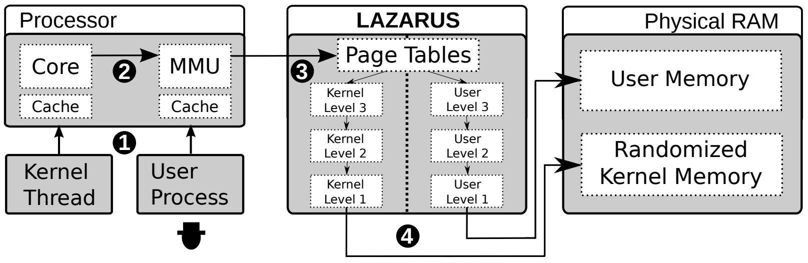

Fig. 2. The idea behind our side channel protection: An unprivileged user process (❶) can exploit the timing side channel for kernel addresses through shared cache access in the MMU paging caches (❷). Our defense mitigates this by enforcing (❸) a separation between different privilege levels for randomized addresses (❹).

enables the attacker to overwrite a code pointer of the kernel to hijack the control-flow of the kernel. However, the attacker cannot use this vulnerability to disclose any addresses.

While modern kernels suffer from software-based information-disclosure vulnerabilities, information-disclosure attacks based on side channels pose a more severe threat because they can be exploited to disclose information in the absence of software vulnerabilities. We address the problem of side channels, and treat software-based information-disclosure vulnerabilities as an orthogonal problem.

3.2 Overview

Usually, kernel and user mode share the same virtual address space. While legitimate accesses to kernel addresses require higher privilege, these addresses still occupy some parts of the virtual memory space that is visible to user processes. The idea behind our side-channel defense is to strictly and efficiently separate randomized kernel memory from virtual memory in user space.

Our idea is depicted in Figure 2. Kernel execution and user space execution usually share a common set of architectural resources, such as the execution unit (Core), and the MMU. The attacker leverages these shared resources in the following way: in step ❶, the attacker sets up the user process and memory setting that will leak the randomization secret. The user process then initiates a virtual memory access to a kernel address.

Next, the processor invokes the MMU to check the required privilege level in step ❷. Since a user space process does not possess the required privileges to access kernel memory, any such access will ultimately be denied. However, to deny access the MMU has to look up the required privileges in the page tables. These are structured hierarchically with multiple levels, and separate caches on every level. Hence, even denied accesses constitute a timing side-channel that directly depends on the last cached level.

We address ❸ the root of this side channel: we separate the page tables for kernel and user space. This effectively prevents side-channel information from kernel addresses to be leaked to user space, because the MMU uses a different page table hierarchy. Thus, while the processor is in user mode, the MMU will not be able to refer to any information about kernel virtual addresses, as shown in step ❹.

3.3 Challenges for Fine-grained Address Space Isolation

To enable LAZARUS to separate and isolate both execution domains a number of challenges have to be tackled: first, we must provide a mechanism for switching between kernel and user execution at any point in time without compromising the randomized kernel memory (**C1**). More specifically, while kernel and user space no longer share the randomized parts of privileged virtual memory, the system still has to be able to execute code pages in both execution modes. For this reason, we have to enable switching between kernel and user space. This is challenging, because such a transition can happen either through explicit invocation, such as a system call or an exception, or through hardware events, such as interrupts. As we will show our defense handles both cases securely and efficiently.

Second, we have to prevent the switching mechanism from leaking any side-channel information (**C2**). Unmapping kernel pages is also challenging with respect to side-channel information, i.e., unmapped memory pages still exhibit a timing difference compared to mapped pages. Hence, LAZARUS has to prevent information leakage through probing of unmapped pages.

Third, our approach has to minimize the overhead for running applications to offer a practical defense mechanism (**C3**). Implementing strict separation of address spaces efficiently is involved, since we only separate those parts of the address space that are privileged and randomized. We have to modify only those parts of the page table hierarchy which define translations for randomized addresses.

In the following we explain how our defense meets these challenges.

C1: Kernel-User Transitioning Processor resources are time-shared between processes and the operating system. Thus, the kernel eventually takes control over these resources, either through explicit invocation, or based on a signaling event. Examples for explicit kernel invocations are *system calls* and *exceptions.* These are synchronous events, meaning that the user process generating the event is suspended and waiting for the kernel code handling the event to finish.

On the one hand, after transitioning from user to kernel mode, the event handler code is no longer mapped in virtual memory because it is located in the kernel. Hence, we have to provide a mechanism to restore this mapping when entering kernel execution from user space.

On the other hand, when the system call or exception handler finishes and returns execution to the user space process, we have to erase those mappings again. Otherwise, paging entries might be shared between privilege levels. Since all system calls enter the kernel through a well-defined hardware interface, we can activate and deactivate the corresponding entries by modifying this central entry point.

Transitions between kernel and user space execution can also happen through *interrupts*. A simple example for this type of event is the timer interrupt, which is programmed by the kernel to trigger periodically in fixed intervals. In contrast to system calls or exceptions, interrupts are asynchronously occurring events, which may suspend current kernel or user space execution at any point in time.

Hence, interrupt routines have to store the current process context before handling a pending interrupt. However, interrupts can also occur while the processor executes kernel code. Therefore, we have to distinguish between interrupts during user or kernel execution to only restore and erase the kernel entries upon transitions to and from user space respectively. For this we facilitate the stored state of the interrupted execution context that is saved by the interrupt handler to distinguish privileged from un-privileged contexts.

This enables LAZARUS to still utilize the paging caches for interrupts occuring during kernel execution.

C2: Protecting the Switching Mechanism The code performing the address space switching has to be mapped during user execution. Otherwise, implementing a switching mechanism in the kernel would not be possible, because the processor could never access the corresponding code pages. For this reason, it is necessary to prevent these mapped code pages from leaking any side-channel information. There are two possibilities for achieving this.

First, we can map the switching code with a different offset than the rest of the kernel code section. In this case an adversary would be able to disclose the offset of the switching code, while the actual randomization secret would remain protected.

Second, we can eliminate the timing channel by inserting dummy mappings into the unmapped region. This causes the surrounding addresses to exhibit an identical timing signature compared to the switching code.

Since an adversary would still be able to utilize the switching code to conduct a code-reuse attack in the first case, LAZARUS inserts dummy mappings into the user space page table hierarchy.

C3: Minimizing Performance Penalties Once paging is enabled on a processor, all memory accesses are mediated through the virtual memory subsystem. This means that a page walk is required for every memory access. Since traversing the page table results in high performance penalties, the MMU caches the most prominent address translations in the Translation Lookaside Buffers (TLBs).

LAZARUS removes kernel addresses from the page table hierarchy upon user space execution. Hence, the respective TLB entries need to be invalidated. As a result, subsequent accesses to kernel memory will be slower, once kernel execution is resumed.

To minimize these perfomance penalties, we have to reduce the amount of invalidated TLB entries to a minimum but still enforce a clear separation between kernel and user space addresses. In particular, we only remove those virtual mappings, which fall into the location of a randomized kernel area, such as the kernel code segment.

4 Prototype Implementation

We implemented LAZARUS as a prototype for the Linux kernel, version 4.8 for the 64 bit variant of the x86 architecture. However, the techniques we used are generic and can be applied to all architectures employing multi-level page tables. Our patch consists of around 300 changes to seven files, where most of the code results from initialization. Hence, LAZARUS should be easily portable to other architectures. Next, we will explain our implementation details. It consists of the initialization setup, switching mechanism, and how we minimize performance impact.

4.1 Initialization

We first setup a second set of page tables, which can be used when execution switches to user space. These page tables must not include the randomized portions of the address space that belong to the kernel. However, switching between privileged and unprivileged execution requires some code in the kernel to be mapped upon transitions from user space. We explicitly create dedicated entry points mapped in the user page tables, which point to the required switching routines.

Fixed Mappings Additionally, there are kernel addresses, which are mapped to fixed locations in the top address space ranges. These *fixmap* entries essentially represent an address-based interface: even if the physical address is determined at boot time, their virtual address is fixed at compile time. Some of these addresses are mapped readable to user space, and we have to explicitly add these entries as well.

We setup this second set of page tables only once at boot time, before the first user process is started. Every process then switches to this set of page tables during user execution.

Dummy Mappings As explained in Section 3, one way of protecting the code pages of the switching mechanism is to insert dummy mappings into the user space page table hierarchy. In particular, we create mappings for randomly picked virtual kernel addresses to span the entire code section. We distribute these mappings in 2M intervals to cover all third-level page table entries, which are used to map the code section. Hence, the entire address range which potentially contains the randomized kernel code section will be mapped during user space execution using our randomly created dummy entries.

4.2 System Calls

There is a single entry point in the Linux kernel for system calls, which is called the system call handler. We add an assembly routine to execute immediately after execution enters the system call handler. It switches from the predefined user page tables to the kernel page tables and continues to dispatch the requested system call. We added a second assembly routine shortly before the return of the system call handler to remove the kernel page tables from the page table hierarchy of the process and insert our predefined user page tables.

However, contrary to its single entry, there are multiple exit points for the system call handler. For instance, there is a dedicated error path, and fast and slow paths for regular execution. We instrument all of these exit points to ensure that the kernel page tables are not used during user execution.

4.3 Interrupts

Just like the system call handler, we need to modify the interrupt handler to restore the kernel page tables. However, unlike system calls, interrupts can occur when the processor is in privileged execution mode as well. Thus, to handle interrupts, we need to distinguish both cases. Basically we could look up the current privilege level easily by querying a register. However, this approach provides information about the current execution context, whereas to distinguish the two cases we require the privilege level of the interrupted context.

Fortunately, the processor saves some hardware context information, such as the instruction pointer, stack pointer, and the code segment register before invoking the interrupt handler routine. This means that we can utilize the stored privilege level associated with the previous code segment selector to test the privilege level of the interrupted execution context. We then only restore the kernel page tables if it was a user context.

We still have to handle one exceptional case however: the non-maskable interrupt (NMI). Because NMIs are never maskable, they are handled by a dedicated interrupt handler. Hence, we modify this dedicated NMI handler in the kernel to include our mechanism as well.

4.4 Fine-grained Page Table Switching

As a software-only defense technique, one of the main goals of LAZARUS is to offer practical performance. While separating the entire page table hierarchy between kernel and user mode is tempting, this approach is impractical.

In particular, switching the entire page table hierarchy invalidates all of the cached TLB entries. This means, that the caches are reset every time and can never be utilized after a context switch. For this reason, we only replace those parts of the page table hierarchy, which define virtual memory mappings for randomized addresses. In the case of KASLR, this corresponds to the code section of the kernel. More specifically, the kernel code section is managed by the last of the 512 level 4 entries.

Thus, we replace only this entry during a context switch between privileged and unprivileged execution. As a result, the caches can still be shared between different privilege levels for non-randomized addresses. As we will discuss in Section 5, this does not impact our security guarantees in any way.

5 Evaluation

In this section we evaluate our prototypical implementation for the Linux kernel. First, we show that LAZARUS successfully prevents all of the previously published side-channel attacks. Second, we demonstrate that our defense only incurs negligible performance impact for standard computational workloads.

5.1 Security

Our main goal is to prevent the leakage of the randomization secret in the kernel to an unprivileged process through paging-based side-channel attacks. For this, we separate the page tables for privileged parts of the address space from the unprivileged parts. We ensure that this separation is enforced for randomized addresses to achieve practical performance.

Because all paging-based exploits rely on the timing difference between cached and uncached entries for privileged virtual addresses, we first conduct a series of timing experiments to measure the remaining side channel in the presence of LAZARUS.

In a second step, we execute all previously presented side-channel attacks on a system hardened with LAZARUS to verify the effectiveness of our approach.

Effect of LAZARUS on the timing side-channel To estimate the remaining timing side-channel information we measure the timing difference for privileged virtual addresses. We access each page in the kernel code section at least once and measure the timing using the `rdtscp` instruction. By probing the privileged address space in this way, we collect a timing series of execution cycles for each kernel code page. The results are shown in Figure 3. [3]

The timing side channel is clearly visible for the vanilla KASLR implementation: the start of the actual code section mapping is located around the first visible jump from 160 cycles up to 180 cycles. Given a reference timing for a corresponding kernel image, the attacker can easily calculate the random offset by subtracting the address of the peak from the address in the reference timing.

In contrast to this, the timing of LAZARUS shows a straight line, with a maximum cycle distance of two cycles. In particular, there is no correlation between any addresses and peaks in the timing signal of the hardened kernel. This indicates that our defense approach indeed closes the paging-induced timing

[3] For brevity, we display the addresses on the x-axis as offsets to the start of the code section (i.e., `0xffffffff80000000`). We further corrected the addresses by their random offset, so that both data series can be shown on top of each other.

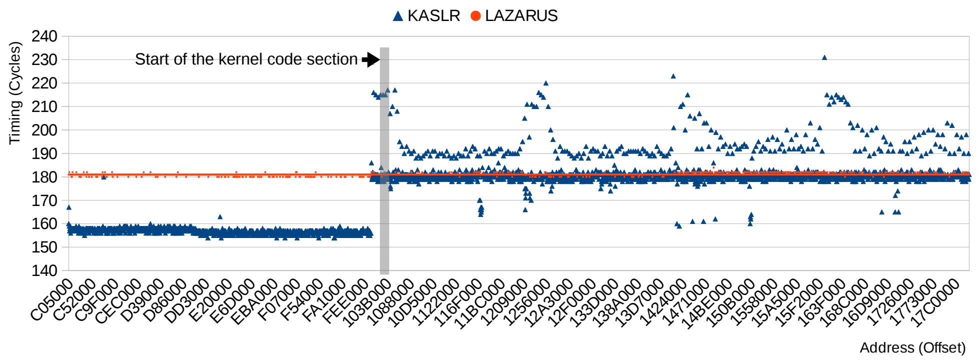

Fig. 3. Timing side-channel measurements.

channel successfully. We note, that the average number of cycles depicted for LAZARUS are also in line with the timings for cached page table entries reported by related work [8, 13]. To further evaluate the security of our approach, we additionally test it against all previous side-channel attacks.

Real-world side-channel attacks We implemented and ran all of the previous side-channel attacks against a system hardened with LAZARUS, to experimentally assess the effectiveness of our approach against real-world attacks.

Page-fault handler The first real-world side-channel attack against KASLR was published by Hund et al. [10]. They noted that the execution time of the page fault handler in the OS kernel depends on the state of the paging caches. More specifically, they access kernel addresses from user space which results in a page fault. While this would usually terminate the process causing the access violation, the POSIX standard allows for processes to handle such events via *signals*. By installing a signal handler for the segmentation violation (`SIGSEGV`), the user process can recover from the fault and measure the timing difference from the initial memory access to the delivery of the signal back to user space. In this way, the entire virtual kernel code section can be scanned and each address associated with its corresponding timing measurement, allowing a user space process to reconstruct the start address of the kernel code section. We implemented and successfully tested the attack against a vanilla Linux kernel with KASLR. In particular, we found that page fault handler exhibits a timing difference of around 30 cycles for mapped and unmapped pages, with an average time of around 2200 cycles. While this represents a rather small difference compared to the other attacks, this is due to the high amount of noise that is caused by the execution path of the page fault handler code in the kernel. [4] When we applied LAZARUS to the kernel the attack no longer succeeded.

[4] This was also noted in the original exploit [10].

Prefetch Recently, the `prefetch` instruction featured on many Intel x86 processors was shown to enable side-channel attacks against KASLR [8]. It is intended to provide a benign way of instrumenting the caches: the programmer (or the compiler) can use the instruction to provide a hint to the processor to cache a given virtual address.

Although there is no guarantee that this hint will influence the caches in any way, the instruction can be used with arbitrary addresses in principle. This means that a user mode program can prefetch a kernel virtual address, and execution of the instruction will fail siltently, i.e., the page fault handler in the kernel will not be executed, and no exception will be raised.

However, the MMU still has to perform a privilege check on the provided virtual address, hence the execution time of the `prefetch` instruction depends directly on the state of the TLBs.

We implemented the prefetch attack against KASLR for Linux, and successfully executed it against a vanilla system to disclose the random offset. Executing the attack against a system hardened with LAZARUS we found the attack to be unsuccessful.

TSX Rafal Wojtczuk originally proposed an attack to bypass KASLR using the Transactional Synchronization Extension (TSX) present in Intel x86 CPUs [23], and the attack gained popularity in the academic community through a paper by Jang et al. [13]. TSX provides a hardware mechanism that aims to simplify the implementation of multi-threaded applications through lock elision. Initially released in Haswell processors, TSX-enabled processors are capable of dynamically determining to serialize threads through lock-protected critical sections if necessary. The processor may abort a TSX transaction if an *atomic* view from the software's perspective is not guaranteed, e.g., due to conflicting accesses between two logical processors on one core.

TSX will suppress any faults that must be exposed to software if they occur within a transactional region. Memory accesses that cause a page walk may abort a transaction, and according to the specification *will not be made architecturally visible through the behavior of structures such as TLBs* [12]. The timing characteristics of the abort, however, can be exploited to reveal the current state of the TLBs. By causing a page walk inside a transactional block, timing information on the aborted transaction discloses the position of kernel pages that are mapped into a process: first, the attacker initiates a memory access to kernel pages inside a transactional block, which causes (1) a page walk, and (2) a segmentation fault. Since TSX masks the segmentation fault in hardware, the kernel is never made aware of the event and the CPU executes the abort handler provided by the attacker-controlled application that initiated the malicious transaction. Second, the attacker records timing information about the abort-handler execution. A transaction abort takes about 175 cycles if the probed page is mapped, whereas it aborts in about 200 cycles or more if unmapped [23]. By probing all possible locations for the start of the kernel code section, this side channel exposes the KASLR offset to the unprivileged attacker in user space.

Probing pages in this way under LAZARUS reveals no information, since we unmap all kernel code pages from the process, rendering the timing side channel useless as any probes to kernel addresses show as unmapped. Only the switching code and the surrounding dummy entries are mapped. However, these show identical timing information, and hence, are indistinguishable for an adversary.

5.2 Performance

We evaluated LAZARUS on a machine with an Intel Core i7-6820HQ CPU clocked at 2.70GHz and 16GB of memory. The machine runs a current release of Arch Linux with kernel version 4.8.14. For our testing, we enabled KASLR in the Linux kernel that Arch Linux ships. We also compiled a secondary kernel with the same configuration and LAZARUS applied.

We first examine the performance overhead with respect to the industry standard SPEC2006 benchmark [9]. We ran both the integer and floating point benchmarks in our test platform under the stock kernel with KASLR enabled. We collected these results and performed the test again under the LAZARUS kernel. Our results are shown in Figure 4.

The observed performance overhead can be attributed to measurement inaccuracies. Our computed worst case overhead is of 0.943%. We should note that SPEC2006 is meant to test computational workloads and performs little in terms of context switching.

To better gauge the effects of LAZARUS on the system, we ran the system benchmarks provided by LMBench3 [22]. LMBench3 improves on the context switching benchmarks by eliminating some of the issues present in previous versions of the benchmark, albeit it still suffers issues with multiprocessor machines. For this reason, we disabled SMP during our testing. Our results are presented in Figure 5.

We can see how a system call intensive application is affected the most under LAZARUS. This is to be expected, as the page tables belonging to the kernel must be remapped upon entering kernel execution. In general, we show a 47% performance overhead when running these benchmarks. We would like to remind the reader, however, that these benchmarks are meant to stress test the performance of the operating system when handling interrupts and do not reflect normal system operation.

In order to get a more realistic estimate of LAZARUS, we ran the Phoronix Test Suite [15], which is widely used to compare the performance of operating systems. The Phoronix benchmarking suite features a large number of tests which cover different aspects of a system, and are grouped according to the targeted subsystem of the machine. Specifically, we ran the system and disk benchmarks to test application performance. Our results are shown in Figure 6. We show an average performance overhead of 2.1% on this benchmark, which is in line with our results provided by the SPEC and LMBench benchmarks. The worst performers were benchmarks that are bound to read operations. We speculate that this is due to the amount of context switches that happen while the read

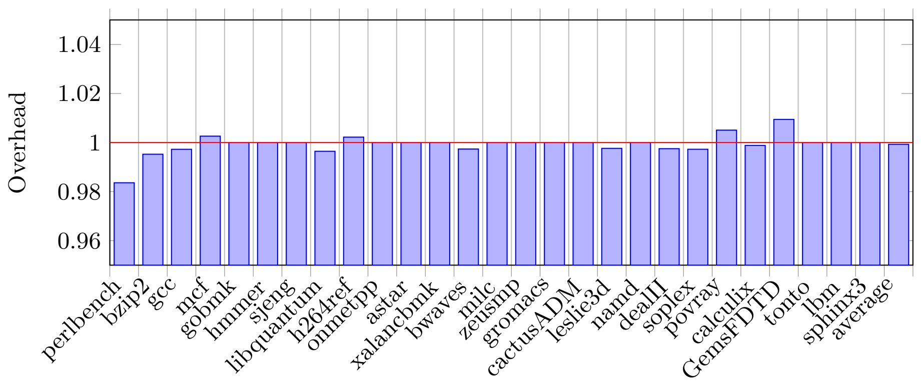

Fig. 4. SPEC2006 Benchmark Results

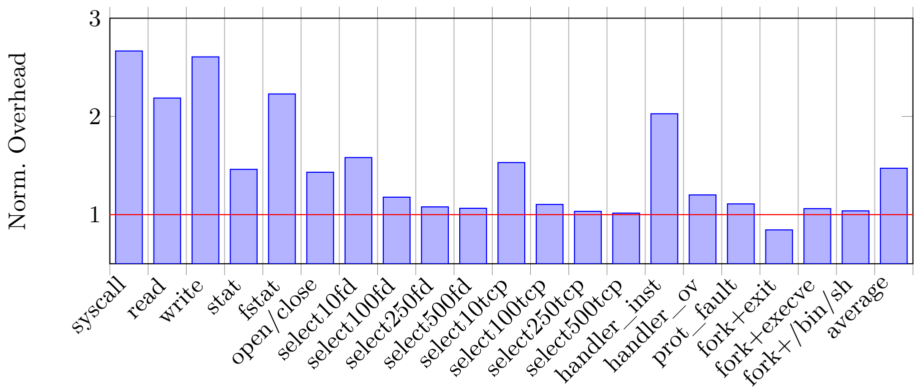

Fig. 5. LMBench3 Benchmark Results

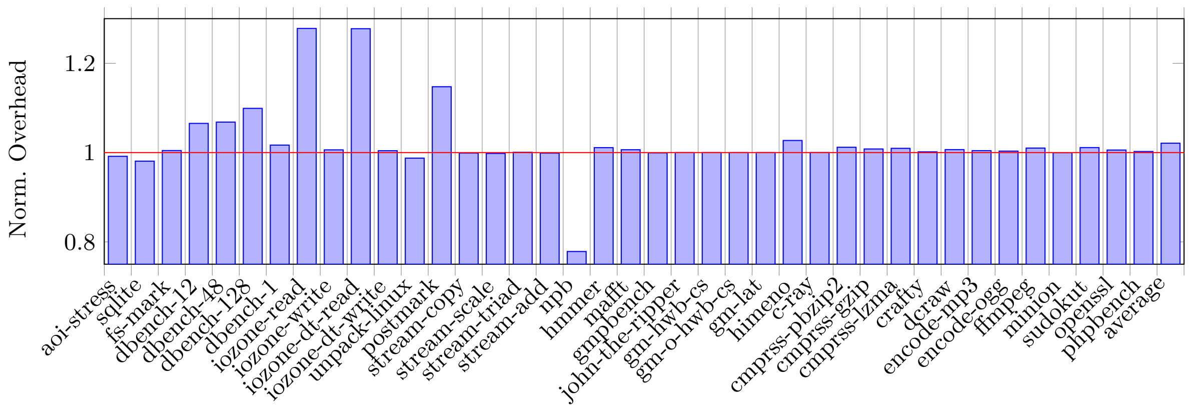

Fig. 6. Phoronix Benchmark Suite

operation is taking place, as a buffer in kernel memory needs to be copied into a buffer from user space or remapped there.

Lastly, we ran the `pgbench` benchmark on a test PostgreSQL database and measured a performance overhead of 2.386%.

6 Discussion

6.1 Applying LAZARUS to different KASLR implementations

Relocation of kernel code is an example of how randomization approaches can be used as a defense building block which is implemented by practically all real-world operating systems [2, 11, 14]. While a kernel employing control-flow integrity (CFI) [1, 3, 21] does not gain security benefit from randomizing the code section, it might still randomize the memory layout of other kernel memory regions: for instance, it can be applied to the module section, to hide the start address of the code of dynamically loadable kernel modules. Further, randomization was recently proposed as a means to protect the page tables against malicious modification through data-only attacks [5].

Since all of the publicly available attacks focus on disclosing the random offset of the kernel code section, we implemented our proof of concept for KASLR as well. Nonetheless, we note that LAZARUS is not limited to hardening kernel code randomization, but can be applied to other randomization implementations as well. In contrast to the case of protecting KASLR, our defense does not require any special treatment for hiding the low-level switching code if applied to other memory regions.

6.2 Other side-channel attacks on KASLR

As explained in Section 2, almost all previously presented side-channel attacks on KASLR exploit the paging subsystem. LAZARUS isolates kernel virtual memory from user processes by separating their page tables. However, Evtyushkin et al. [6] recently presented the branch target buffer (BTB) side-channel attack, which does not exploit the paging subsystem for virtual kernel addresses.

In particular, they demonstrated how to exploit collisions between branch targets for user and kernel addresses. The attack works by constructing a malicious chain of branch targets in user space, to fill up the BTB, and then executing a previously chosen kernel code path. This evicts branch targets previously executed in kernel mode from the BTB, thus their subsequent execution will take longer.

While the BTB attack was shown to bypass KASLR on Linux, it differs from the paging-based side channels by making a series of assumptions: 1) the BTB has a limited capacity of 10 bits, hence it requires KASLR implementations to deploy a low amount of entropy in order to succeed. 2) it requires the attacker to craft a chain of branch targets, which cause kernel addresses to be evicted from the BTB. For this an adversary needs to reverse engineer the hashing algorithm used to index the BTB. These hashing algorithms are different for every micro architecture, which limits the potential set of targets. 3) the result of the attack can be ambiguous, because any change in the execution path directly effects the BTB contents.

There are multiple ways of mitigating the BTB side-channel attack against KASLR. A straightforward approach is to increase the amount of entropy for KASLR, as noted by Evtyushkin et al. [6]. A more general approach would be to introduce a separation between privileged an unprivileged addresses in the BTB. This could be achieved by offering a dedicated flush operation, however this requires changes to the hardware. Alternatively, this flush operation can emulated in software, if the hashing algorithm used for indexing the BTB has been reverse engineered. We implemented this approach against the BTB attack by calling a function which performs a series of jump instructions along with our page tables switching routine and were unable to recover the correct randomization offset through the BTB attack in our tests.

7 Related Work

In this section we discuss software and hardware mitigations against side-channel attacks that were proposed, and compare them to our approach.

7.1 Hardware Mitigations

Privilege Level Isolation in the Caches Eliminating the paging side channel is also possible by modifying the underlying hardware cache implementation. This was first noted by Hund et al. [10]. However, modern architectures organize caches to be optimized for performance. Additionally, changes to the hardware are very costly, and it takes many years to widely deploy these new systems. Hence, it is unlikely that such a change will be implemented, and even if it is, existing production systems will remain vulnerable for a long time. Our software-only mitigation can be deployed instantly by patching the kernel.

Disabling Detailed Timing for Unprivileged Users All previously presented paging side-channel attacks rely on detailed timing functionality, which is provided to unprivileged users by default. For this reason, Hund et al. [10] suggested to disable the `rdtsc` instruction for user mode processes. While this can be done from software, it effectively changes the ABI of the machine. Since modern platforms offer support for a large body of legacy software, implementing such a change would introduce problems for many real-world user applications. As we demonstrate in our extensive evaluation, LAZARUS is transparent to user-level programs and does not disrupt the usual workflow of legacy software.

7.2 Software Mitigations

Separating Address Spaces Unmapping the kernel page tables during user-land execution is a natural way of separating their respective address spaces, as suggested in [8, 13]. However, Jang et al. [13] considered the approach impractical, due to the expected performance degradation. Gruss et al. [8] estimated the performance impact of reloading the entire page table hierarchy up to 5%, by reloading the top level of the page table hierarchy (via the `CR3` register) during a context switch, but did not provide any implementation or detailed evaluation of their estimated approach. Reloading the top level of the page tables results in a higher performance overhead, because it requires the processor to flush all of the cached entries. Address space separation has been implemented by Apple for their iOS platform [16]. Because the ARM platform supports multiple sets of page table hierarchies, the implementation is straightforward on mobile devices. For the first time we provide an improved and highly practical method of implementing address space separation on the x86 platform.

Increasing KASLR Entropy Some of the presented side-channel attacks benefit from the fact that the KASLR implementation in the Linux kernel suffers from a relatively low entropy [6, 10]. Thus, increasing the amount of entropy represent a way of mitigating those attacks in practice. While this approach was suggested by Hund et al. [10] and Evtyushkin et al. [6], it does not eliminate the side channel. Additionally, the mitigating effect is limited to attacks which exploit low entropy randomization. In contrast, LAZARUS mitigates all previously presented paging side-channel attacks.

Modifying the Page Fault Handler Hund et al. [10] exploited the timing difference through invoking the page fault handler. They suggested to enforce its execution time to an equal timing for all kernel addresses through software. However, this approach is ineffective against attacks which do not invoke the kernel [8, 13]. Our mitigation reorganizes the cache layout in software to successfully stop the attacks, that exploit hardware features to leak side channel information, even for attacks that do not rely on the execution time of any software.

KAISER Concurrently to our work Gruss et al. implemented strong address-space separation [7]. Their performance numbers are in line with our own measurements, confirming that separating the address spaces of kernel and userland constitutes a practical defense against paging-based side-channel attacks. In contrast to LAZARUS, their approach does not make use of dummy mappings to hide the switching code, but separates it from the rest of the kernel code section (as outlined in 3.3.C2).

8 Conclusion

Randomization has become a vital part of the security architecture of modern operating systems. Side-channel attacks threaten to bypass randomization-based defenses deployed in the kernel by disclosing the randomization secret from unprivileged user processes. Since these attacks exploit micro-architectural implementation details of the underlying hardware, closing this side channel through a software-only mitigation efficiently is challenging. However, all of these attacks rely on the fact that kernel and user virtual memory reside in a shared address space. With LAZARUS, we present a defense to mitigate previously presented side-channel attacks purely in software. Our approach shows that side-channel information exposed through shared hardware resources can be hidden by separating the page table entries for randomized privileged addresses from entries for unprivileged addresses in software. LAZARUS is a necessary and highly practical extension to harden kernel-space randomization against side-channel attacks.

Acknowledgment

This work was supported in part by the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union's Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP.

Dean Sullivan, Orlando Arias, and Yier Jin are partially supported by the Department of Energy through the Early Career Award (DE-SC0016180). Mr. Orlando Arias is also supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 1144246.

Bibliography

[1] Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Transactions on Information System Security 13 (2009)

[2] Cook, K.: Kernel address space layout randomization. `http://selinuxproject.org/~jmorris/lss2013_slides/cook_kaslr.pdf` (2013)

[3] Criswell, J., Dautenhahn, N., Adve, V.: Kcofi: Complete control-flow integrity for commodity operating system kernels. In: 35th IEEE Symposium on Security and Privacy. S&P (2014)

[4] CVEDetails: CVE-2016-4557. `http://www.cvedetails.com/cve/cve-2016-4557` (2016)

[5] Davi, L., Gens, D., Liebchen, C., Ahmad-Reza, S.: PT-Rand: Practical mitigation of data-only attacks against page tables. In: 24th Annual Network and Distributed System Security Symposium. NDSS (2017)

[6] Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Jump over aslr: Attacking branch predictors to bypass aslr. In: IEEE/ACM International Symposium on Microarchitecture (MICRO) (2016)

[7] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S.: Kaslr is dead: Long live kaslr. In: International Symposium on Engineering Secure Software and Systems. ESSoS (2017)

[8] Gruss, D., Maurice, C., Fogh, A., Lipp, M., Mangard, S.: Prefetch side-channel attacks: Bypassing smap and kernel aslr. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 368–379. ACM (2016)

[9] Henning, J.L.: Spec cpu2006 benchmark descriptions. SIGARCH Comput. Archit. News 34(4), 1–17 (Sep 2006), http://doi.acm.org/10.1145/1186736.1186737

[10] Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: 34th IEEE Symposium on Security and Privacy. S&P (2013)

[11] Inc., A.: Os x mountain lion core technologies overview. http://movies.apple.com/media/us/osx/2012/docs/OSX_MountainLion_Core_Technologies_Overview.pdf (2012)

[12] Intel: Intel 64 and IA-32 architectures software developer's manual. http://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (2017)

[13] Jang, Y., Lee, S., Kim, T.: Breaking kernel address space layout randomization with intel TSX. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 380–392. ACM (2016)

[14] Johnson, K., Miller, M.: Exploit mitigation improvements in windows 8. https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf (2012)

[15] Larabel, M., Tippett, M.: Phoronix test suite. h ttp://www. phoronix-test-suite. com (2011)

[16] Mandt, T.: Attacking the ios kernel: A look at "evasi0n". http://www.nislab.no/content/download/38610/481190/file/NISlecture201303.pdf (2013)

[17] MITRE: CVE-2015-1328. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1328 (2015)

[18] MITRE: CVE-2016-0728. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0728 (2016)

[19] MITRE: CVE-2016-5195. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195 (2016)

[20] Molinyawe, M., Hariri, A.A., Spelman, J.: $hell on earth: From browser to system compromise. In: Blackhat USA. BH US (2016)

[21] PaX Team: RAP: RIP ROP (2015)

[22] Staelin, C.: lmbench: an extensible micro-benchmark suite. Software-Practice and Experience 35(11), 1079 (2005)

[23] Wojtczuk, R.: Tsx improves timing attacks against kaslr. https://labs.bromium.com/2014/10/27/tsx-improves-timing-attacks-against-kaslr/ (2014)

ATRIUM: Runtime Attestation Resilient Under Memory Attacks

Shaza Zeitouni
TU Darmstadt, Germany
shaza.zeitouni@trust.tu-darmstadt.de

Ghada Dessouky
TU Darmstadt, Germany
ghada.dessouky@trust.tu-darmstadt.de

Orlando Arias
University of Central Florida, USA
oarias@knights.ucf.edu

Dean Sullivan
University of Florida, USA
deanms@ufl.edu

Ahmad Ibrahim
TU Darmstadt, Germany
ahmad.ibrahim@trust.tu-darmstadt.de

Yier Jin
University of Florida, USA
yier.jin@ece.ufl.edu

Ahmad-Reza Sadeghi
TU Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

***Abstract*—Remote attestation is an important security service that allows a trusted party (verifier) to verify the integrity of a software running on a remote and potentially compromised device (prover). The security of existing remote attestation schemes relies on the assumption that attacks are *software-only* and that the prover's code cannot be modified at runtime. However, in practice, these schemes can be bypassed in a stronger and more realistic adversary model that is hereby capable of controlling and modifying code memory to attest benign code but execute malicious code instead – leaving the underlying system vulnerable to Time of Check Time of Use (TOCTOU) attacks.**

In this work, we first demonstrate TOCTOU attacks on recently proposed attestation schemes by exploiting physical access to prover's memory. Then we present the design and proof-of-concept implementation of ATRIUM, a *runtime* remote attestation system that securely attests both the code's binary and its execution behavior under memory attacks. ATRIUM provides resilience against both software- and hardware-based TOCTOU attacks, while incurring minimal area and performance overhead.

***Index Terms*—Attestation, runtime, memory attacks**

I. Introduction

Recent high-profile attacks on embedded systems, such as Mirai and Stuxnet, have become crucially alarming and of increased significance as systems are becoming more interconnected and collaborative. *Remote attestation* plays an important role as a security service for detecting malware on a remote device. It is implemented as a challenge-response protocol that allows a trusted *verifier* to obtain an authentic report about the (software) state of a potentially untrusted remote device called *prover*. Conventional attestation schemes are static in nature, i.e., the prover sends an authenticated report to the verifier by issuing a digital signature or cryptographic MAC (Message Authentication Code) over the verifier's challenge and the *measurement* (typically hash) of the binary code to be attested [22]. However, static attestation only ensures the integrity of binaries but *not* of their execution. In particular, it cannot detect the prevalent state-of-the-art runtime attacks that do not modify the program binary but subvert the intended control flow of the targeted application program during its execution. Current runtime attacks take advantage of code-reuse techniques, such as return-oriented programming that dynamically generate malicious code by chaining together code snippets (called gadgets) of benign code *without* requiring to inject any malicious code/instructions [24]. Consequently, the hash value computed over the binaries remain unchanged and the attestation protocol succeeds, although the prover has been compromised. These sophisticated exploitation techniques have been shown effective on many processor architectures, such as Intel x86 [23], SPARC [4], ARM [16], and Atmel AVR [10]. In fact, large-scale investigations of embedded systems security have shown various vulnerabilities, including memory corruption (such as buffer overflow) that can be exploited for runtime attacks.

Hence, effective attestation should enable reporting the prover's dynamic behavior – more concretely, its current execution details – to the verifier. To attest the dynamic program behavior researchers have proposed enhancements and/or extensions to static binary attestation (e.g., [11], [3]). The most recent, C-FLAT [3], reports the prover's dynamic state (execution paths) and provides fine-grained control-flow measurements to the verifier. Note that, unlike control-flow integrity (CFI) enforcement, control-flow attestation provides detailed information about the executed path that might be of crucial interest to a remote verifier. This information helps in detecting data-oriented non-control attacks [5] that can bypass CFI by corrupting data variables to execute a valid but unintended control-flow path, for instance, redirecting the control flow to a high-privileged recovery routine (see also [13]). However, C-FLAT requires program code instrumentation and incurs high performance overhead, particularly on the prover.

On the other hand, all existing attestation schemes (including C-FLAT) rule out physical attacks in their adversary model. This assumption is not always realistic, since the adversary may at some point have physical access to the prover. In this case, it is possible to execute (extraordinarily effective and cheap) non-invasive attacks on the program code memory through *physical access*. In particular, the adversary physically controls and modifies the memory such that benign code is attested but malicious code is executed instead.

Goals and Contributions. In this paper, we first demonstrate that – using external interfacing with prover's program code memory bank – an adversary can bypass all existing attestation schemes and deliver sound attestation reports, without even having to extract the prover's secret keys (cf. § III). To overcome the limitations of current attestation schemes, we introduce a holistic approach to attestation ATRIUM, a *resilient runtime attestation* scheme that is capable of detecting both physical memory attacks and software attacks including runtime attacks by attesting the executed instructions and their control flow at runtime. Our main contributions are listed as follows.

- We demonstrate memory bank attacks on state-of-the-art attestation schemes for embedded devices such as SMART [9] and C-FLAT [3]. We exploit physical access to code memory to bypass attestation and deliver sound attestation reports without having to extract the prover's secret keys.
- We present ATRIUM– an attestation scheme which: (1) detects memory bank attacks by attesting instructions as they are fetched from (off-chip) memory for execution; (2) prevents software attacks on the attestation process itself by separating the attestation engine from the processor (i.e., no instructions are sent to the processor to perform attestation). Instead, attestation is performed by a separate hardware engine in parallel. (3) detects runtime attacks by tracking and reporting both executed instructions and control-flow events during execution.
- We present a proof-of-concept implementation and performance analysis which demonstrate the effectiveness and feasibility of ATRIUM, and its applicability to low-end embedded devices.

II. Background

Control-Flow Graph (CFG). The execution flow of a program can be abstracted into a control-flow graph (CFG) by leveraging the aid of static and dynamic code analysis. The nodes in CFG represents basic blocks of a program, while edges represent control-flow transitions from one block to another by means of a branch instruction. A *valid* path in CFG is composed of several nodes connected by edges.

Runtime Attacks. An outline of the different classes of runtime attacks is illustrated in Figure 1. The system dedicates separate memories for data and code. The former is marked as readable and writable *(rw)*, while the latter is marked as readable and executable *(rx)*. This ensures that code cannot be executed from data memory, and code memory cannot be overwritten *by means of software*. Along this CFG, we can outline three major classes of runtime attacks: ❶ non-control-data attacks that indirectly affect the control flow of a program, ❷ corruption of loop variables, and ❸ code-pointer overwrite attacks. By corrupting control-flow information stored in the stack or heap and overwriting code-pointers (return addresses and function pointers) as in ❸ an attacker can redirect the control flow of a program such that execution has a malicious and unauthorized effect. In attacks based on *code-injection*,

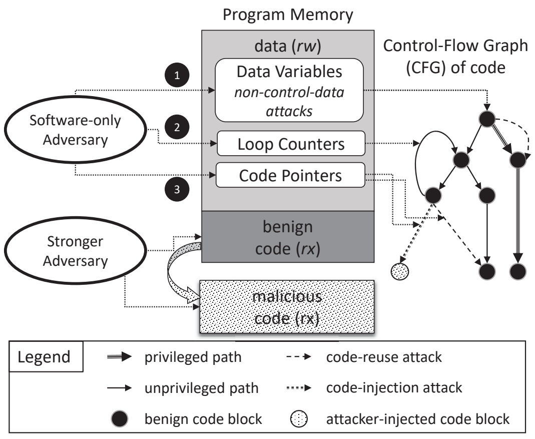

Figure 1: Different attack classes

the attacker places a malicious executable payload in program memory and redirects control flow to execute it. Alternatively, state-of-the-art runtime attacks leverage *code-reuse* techniques, such as *Return-oriented Programming* (ROP) [23]. These attacks exploit a memory corruption vulnerabilities (e.g., buffer overflows) in the program and stitch together a malicious sequence of machine code instructions from benign *gadgets* of code already residing in the memory of the vulnerable program. *Non-control-data attacks* [5] do not compromise the control flow of a program, but cause unexpected malicious control flow by corrupting critical data variables such as an authentication variable. This results in executing a privileged (unintended) but permissible control-flow path that exists in the CFG. Attack ❷ affects the number of times a program loop executes by corrupting a loop variable such as a counter. This can have severe consequences depending on the context, e.g., a syringe pump dispenses more liquid than requested (see [3]). Code injection attacks can be prevented by either marking memory as writable or executable. This mechanism is known as *Data Execution Prevention* (DEP) [12]. Countermeasures against code reuse attacks include: *Control-Flow Integrity* (CFI) [2], fine-grained code randomization [19], and Code-Pointer Integrity (CPI) [18].

Besides software-based runtime attacks, a stronger adversary as shown in Figure 1, can modify program code in memory through *physical access* without mounting sophisticated invasive physical attacks, but by simply replacing the benign code memory with malicious code memory at runtime. We elaborate on these memory bank attacks next in § III and propose an attestation scheme that can mitigate them in § V.

III. TOCTOU Attacks on Attestation Schemes

Next we describe memory bank attacks that we aim to mitigate in this work, and we show how they bypass recently proposed attestation schemes: SMART [9] C-FLAT [3] and LO-FAT [7]. These attacks assume a stronger adversary that can physically manipulate the code memory without the need for sophisticated invasive physical attacks and can consequently bypass attestation schemes that strictly consider software-only adversary. The attack is illustrated in Figure 2: At $\mathcal{P}rv$'s side the attestation scheme (i.e., the attestation code and secret key) is stored on-chip while the benign code resides in an external memory. The adversary can interleave instruction fetches to malicious code in-between those fetches needed to attest the benign code of the original program. This can be done by replacing the original memory interface with an interface to a memory controller. This allows the adversary to direct instruction fetches to either benign code when attestation is running, or malicious code otherwise. The same interleaving attack can be achieved by inserting malicious instructions in-between hooks to the attestation. As long as the malicious instructions do not interfere with attesting benign code, e.g., intended control flow, the attestation can be bypassed. In the following, we describe how we implement the attacks to bypass SMART and C-FLAT.

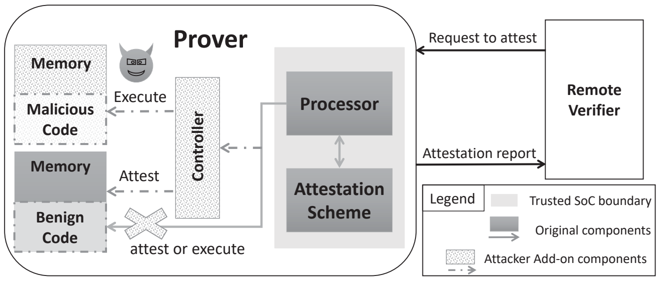

Figure 2: Memory bank attack on attestation schemes

A. SMART

SMART [9] is a static attestation scheme that establishes a root of trust in low-end embedded systems with minimal hardware components. It targets microprocessors that are able to execute code from an external memory, whereas the attestation code and key reside in an internal ROM and are protected by access control policies of a memory protection unit (MPU). When an attestation request is received, the *atomic* attestation code in ROM computes a HMAC of a region of code memory, provided in the attestation request. Then the attested code executes *atomically*.

Detecting Attestation Execution. By eavesdropping in the communication channel between the verifier and the prover for an attestation request, we determine when the attestation engine is about to run in order to launch a TOCTOU attack. Although this is permissible by the adversary model in SMART, we choose not to tackle the detection problem this way. Instead, we examine a side-channel that is inherent to the SMART design, by placing a monitor on the address bus between the processor and memory to capture which addresses are being accessed. Using the access patterns, we are able to discern whether a CPU is executing from external memory or from the internal ROM. Since SMART is prototyped on the open-source MSP430[1], it utilizes a von Neumann architecture, where data and instructions are accessed over the same address space but are structured such that they reside in different sections of memory. Hence, we can extract and filter out data accesses, leaving behind accesses to code memory. In doing so, we observe the time-frame that it takes the internal ROM to set up the attestation environment, followed by the linear scan of code addresses, then the subsequent execution of external code. On processors with modified Harvard architecture, a temporary halt in accesses to code memory would be recognized, as the ROM code starts executing. We then observe a linear scan over an address range, as code is being read and hashed by the attestation code. A break is then noticed as the ROM code cleans up memory, followed by the continued access to program memory for execution. Utilizing this, we perform one of the following attacks to mount a TOCTOU attack.

Blind Execution of Malicious Software. Since code memory remains external to the SoC, we splice the address bus, add a new memory chip containing malicious code and utilize the monitor to detect when the attestation code runs. When attesting, we bank to the memory with the intended code. When executing, we bank to the malicious code memory, allowing SMART to report valid attestation results while malicious code is actually executed by CPU during periods of no attestation.

Leakage of Secrets via Data Memory Banking. As the attestation code runs, temporary values are saved in memory, assuming SMART implementation utilizes off-chip memory to store temporary values. We use the monitor to detect when the attestation code runs. As data memory is accessed to store temporary values, we bank memories to allow for the leakage of values. We perform this by physically tampering with the address lines between the processor and the memory. As the monitor detects when SMART is about to perform its cleanup routines, we bank to a different portion of memory, leaving the ROM code to erase the wrong portion of memory. By reading the SMART secrets from memory, we are able to reconstruct the attestation secret and fake a valid response.

B. C-FLAT

C-FLAT [3] is a runtime attestation scheme that aims to measure and report the control-flow behavior of an executing code. It instruments all branch instructions such that they are intercepted by a *runtime tracer* (RTT). The RTT recovers the source and destination addresses of the branch as well as its type, which are then passed to the *measurement engine* (ME). The ME is responsible for computing a hash over the reported branches and these hash measurements are secured by running in a TrustZone secure world. In this way, a runtime control-flow attestation report is generated and verified against previously computed control-flow traces stored in a trusted verifier party.

C-FLAT is susceptible to two TOCTOU attacks assuming that the attacker has physical access to the code memory : 1) replacing instructions within a basic block with malicious ones; and 2) refactoring the control-flow graph (CFG) of an arbitrary program to match a benign CFG protected by C-FLAT. Both attacks exploit the fact that C-FLAT attests *only* control flow when exiting a basic block but not the executed instructions themselves. Hence, intermediate instructions within the basic block can be arbitrarily replaced by malicious executable code by a stronger adversary with physical access to the code

[1]http://opencores.org/project,openmsp430

memory, as long as the control flow of the code remains unchanged and the expected attestation report is not violated. These attacks are also applicable to the hardware-assisted control-flow attestation scheme LO-FAT [7] since it also only attests control flow.

We chose to implement a TOCTOU attack against one of the case studies presented in [3], namely the syringe pump program responsible for dispensing intravenous (IV) fluids. Our attack goal is to dispense liquid in incorrect volumes at unexpected times, thereby, disrupting the correct flow of IV fluids. We only demonstrate the second attack variant, however, the first variant of the attack is also easily feasible by replacing the original instructions within the basic block with malicious ones. This allows the original RTT hooks into the ME to compute a valid attestation report as it is based upon the source and destination addresses of a branch and its type.

In place of the original program that manages liquid dispensing and withdrawal, we implement a malicious program that chooses a random value to dispense by modifying the `set-quantity` function and additionally creates compound dispense and withdraw triggers for the `move-syringe` function. We embed this code in the original program, which creates new edges in the CFG of the syringe pump program. Our new edges would violate C-FLAT's attestation report for the benign syringe pump program.

To avoid triggering C-FLAT, we refactor the CFG of our attacker syringe pump program using the REpsych tool[2] to construct the desired CFG. The REpsych tool is an IDA plugin that translates a source image into a functioning program whose CFG is the image. We used the original syringe pump's CFG as a source image, and our modified syringe pump program as the target. This allowed us to generate a program with alternative functionality, but equivalent CFG to the original syringe pump program. We then recompute the attestation report using C-FLAT's tools[3]. The attacker program's attestation report matched the original syringe pump program's attestation report after CFG refactoring. Thus, we were able to accurately execute the attacker program without violating C-FLAT's protection.

IV. ATRIUM

We present ATRIUM a runtime attestation scheme targeting bare-metal embedded systems software. ATRIUM comprises a remote embedded system, called in this context the prover $\mathcal{P}rv$, and a trusted verifier $\mathcal{V}rf$. The $\mathcal{P}rv$ is deployed in-field such that the adversary has physical access to its memory. Typically, both $\mathcal{V}rf$ and $\mathcal{P}rv$ have access to the binary code of the program P to be attested on $\mathcal{P}rv$. Note that, in practice, it may not be feasible to apply runtime attestation to the entire program code due to obvious efficiency reasons, but it can be applied to pre-defined security-critical code regions.

A. Adversary Model and Assumptions

In addition to the standard capabilities of the adversary in typical remote attestation schemes, which assume software-only attacks, our adversary can also perform runtime attacks (§ II). Furthermore, we assume a stronger adversary that has physical access to the $\mathcal{P}rv$'s memory and can manipulate the program code at runtime and, therefore, is able to mount a TOCTOU attack (§ III). However, the adversary cannot modify memory reserved and used by ATRIUM itself – this memory is hardware-protected and not mapped to the software-accessible address space. *Data-oriented programming attacks* [13] that do not affect the control flow as well as invasive physical attacks on the SoC that aim at extracting secret keys are out of scope. This assumption is reasonable, since an adversary is more likely to mount a simple physical attack on the memory as we demonstrated in § III, rather than expensive sophisticated invasive attacks on the chip that can destruct it eventually.

B. Runtime Attestation: High-Level Scheme

Inspired by C-FLAT [3] (described in § III-B) and the hardware-assisted scheme LO-FAT [7], ATRIUM performs attestation of an executing program code at runtime. However, unlike both schemes, it measures both the executed instructions (to detect the more advanced TOCTOU attacks described in § III) and control flow (to detect runtime attacks).

Similar to C-FLAT, our attestation mechanism relies on $\mathcal{V}rf$ performing one-time offline pre-processing to generate the CFG of program P (including expected loop execution information) by means of static and dynamic analysis. $\mathcal{V}rf$ computes cryptographic hash measurements over the instructions and addresses of basic blocks along legal CFG paths and stores them in a reference database. $\mathcal{V}rf$ initiates the attestation by sending $\mathcal{P}rv$ benign input in_b, the code region to be attested in P, and a nonce to ensure freshness of the attestation report. $\mathcal{P}rv$ executes P on the benign inputs in_b and potentially malicious inputs in_m that are not controlled by $\mathcal{V}rf$ and may lead to the corruption of the program's control flow by means of runtime attacks (§ II). ATRIUM is triggered during the execution of the code region of interest and computes a set of hash measurements over the executed paths. When execution of the code region is complete, $\mathcal{P}rv$ generates and sends to $\mathcal{V}rf$ the final *attestation report* consisting of the concatenated set of hash values $H_0\|...\|H_n$ and the number of iterations of the hash values which correspond to executed loop paths, and a signature over $H_0\|...\|H_n$ and the nonce based on $\mathcal{P}rv$'s secret key sk. To ensure authenticity of the report, sk is stored in memory accessible only by ATRIUM. Upon receiving the report, $\mathcal{V}rf$ verifies its signature using $\mathcal{P}rv$'s public key pk and checks whether the $H_0\|...\|H_n$ values match the reference hash values under input in_b. If they match, $\mathcal{V}rf$ concludes that $\mathcal{P}rv$'s execution of the attested code region was correct in terms of executed instructions and their control flow. For better understanding, we demonstrate next by an example how the hash values are computed during attestation.

Example. A CFG of an example pseudo-code is shown in Figure 3. Each numbered node in the CFG represents the corresponding numbered *basic block* of sequential instructions in the pseudo-code and the address of the first instruction of that basic block. For example, $\mathbf{N}_5$ corresponds to the first 3

[2]https://github.com/xoreaxeaxeax/REpsych
[3]https://github.com/control-flow-attestation/c-flat

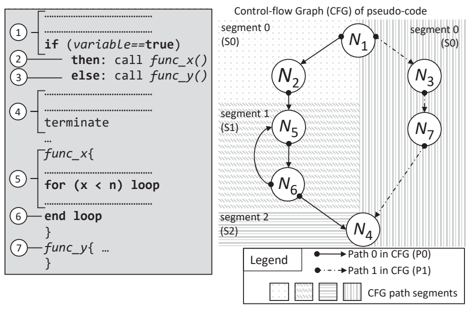
Figure 3: Example pseudo-code and its segmented CFG

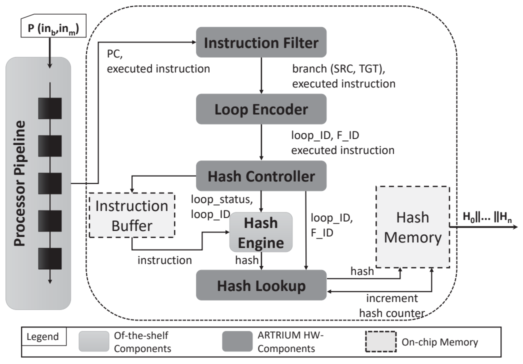
Figure 4: Architecture of ATRIUM

instructions outlined in the pseudo-code, constituting a single basic block, and the address of the first instruction. The CFG shown in Figure 3 has 2 main paths: **P0**, in bold, consisting of nodes N_1-N_2-N_5-N_6-N_4 and **P1**, in dashed, consisting of nodes N_1-N_3-N_7-N_4. In order to avoid combinatorial explosion of legal hash values that would occur due to multiple loop iterations, the program CFG is split into segments such that hash values for loop paths are computed separately, rather than computing a single hash value over the complete executed path of the attested region. In Figure 3, due to the loop in N_5-N_6, **P0** is sectioned into 3 segments: $S0$, $S1$ and $S2$. $S0$ comprises all nodes till loop entry at N_5, where $S1$ is initialized. $S1$ ends at the loop exit node N_6, and $S2$ is initialized at N_4 and beyond until again another loop is encountered and so on.

When path **P0** is executed and attested, ATRIUM accumulates nodes (address of the first instruction and the individual instructions in each node) along each segment and computes a hash value for each segment: a hash value $H_0 = H(N_1||N_2)$ over the nodes in $S0$ of **P0**, followed by $H_1 = H(N_5||N_6)$ over the nodes in $S1$, and $H_2 = H(N_4)$ over the nodes in $S2$, resulting in the set of hash values $H_0||H_1||H_2$ representing the executed path **P0**. **P1**, on the other hand, has no loops. Therefore, when executed the whole path is measured by a single hash value $H_3 = H(N_1||N_3||N_7||N_4)$. This CFG segmentation in hash computation allows our scheme to tackle loops and nested loops efficiently, while also allowing fine-grained attestation of their execution. It requires that ATRIUM can detect and interpret loops accurately at runtime. Unlike C-FLAT, we aim to realize this without instrumentation, hence avoiding the associated performance overheads. We present next the architecture of ATRIUM and how it interfaces directly with the processor hardware to capture at runtime every executed instruction and accurately interpret control flow and infer loop entry and exit points *without instrumentation*.

V. ATRIUM: Design and Implementation

ATRIUM is a hardware-based scheme for runtime attestation that tightly integrates with a processor, as shown in Figure 4. This allows it to extract the executed instructions and their memory addresses from the execute stage of the pipeline at runtime while the program P (that needs to be attested) executes on input values in_b and in_m. ATRIUM outputs a set of hash values $H_0||...||H_n$ computed over the executed path which get included in the attestation report. We present next the components of ATRIUM and their implementation details.

A. Instruction Filter

Upon code execution, the instruction filter extracts the current program counter (PC) and the executed instruction per clock cycle and checks whether the current instruction is a branch or jump, since such instructions reflect control-flow transitions.

Implementation. We implemented the instruction filter such that it tightly extends the execute stage of the processor from which it extracts the PC and instruction per clock cycle. If the current instruction is a control-flow instruction, its PC and the address it jumps to are stored as source–target pair, (Src, Tgt)-pair. To determine whether the branch was taken and whether control jumped forwards or backwards in memory, the PC of the next executed instruction is compared to the stored target address. Instruction filter outputs the following signals: (1) branch instructions, their type, and (Src, Tgt)-pairs and (2) basic block addresses and executed instructions.

B. Loop Encoder

As explained in § IV-B, ATRIUM handles loops and their hash computations differently. Hence, at runtime the loop encoder detects loops and identifies their entry and exit points and their depth, in case of nested loops. It checks whether the behavior of a captured branch can be inferred as returning to a loop's entry point, hence indicating a new loop iteration. The loop encoder instructs the hash controller to finalize the ongoing hash computation and initialize a new one with the entry address of a loop iteration. Furthermore, the loop encoder also detects if a branch represents a system call since system functions have to be handled specially in ATRIUM.

Implementation. To detect loops at runtime without relying on code instrumentation, we utilize a feature of RISC architectures that implement a *link-register*, such as PowerPC, ARM, SPARC, and RISC-V. We adopt a heuristic used in [7] to distinguish between backward branches that indicate loop entry, and branches for subroutine calls where the call target resides earlier in memory. Subroutine calls use instructions that update the *link-register* with the return address, hence, we consider any *non-linking* backwards branch as a *loop entry node*. Consequently, the basic block after the branch instruction is considered a *loop exit node*. This is based on observations of the RISC-V compiler assembly and its calling convention: any subroutine call with multiple call sites must be *linking* and updates the *link-register*. Subroutines with a single call site can be compiled as a *linking* branch or inlined using the RISC-V compiler. A system call is identified by comparing its target against a predefined list of addresses of such functions and issuing a unique identifier for that function F_ID. The loop encoder stores the addresses of entry and exit nodes of each loop in a content-addressable memory (CAM) to ensure single-cycle constant-access search time. At runtime, every (Src, Tgt) is used to index the CAM to detect if a loop is re-entered or exits and to extract its loop_ID and depth (in case of nested loops). An iterations counter for each loop is maintained and updated at runtime. We detect loop exit when execution proceeds past the currently active loop exit node, either due to sequential execution or a non-linking jump, such as a *break*. The F_ID, loop_ID and loop_status signals are forwarded then to the hash controller.

C. Hash Engine and Hash Controller

The hash engine computes a hash value of each executed path within a segment (§ IV-B). The hash controller regulates the operation of the hash engine, i.e., finalizes or initiates a hash computation based on the control signals received from the loop encoder. In case the computed hash corresponds to a loop path, the hash controller sends this hash to the hash lookup and sets the search boundaries of the hash lookup to that particular current loop (necessary in case of nested loops). Otherwise, the hash value is simply stored in hash memory.

Implementation. We selected Blake2 [4] for hash computations and used the open-source hardware implementation of Blake2b, which takes as an input a message block of size 1Kbit and has a configurable digest size. We configured its digest size to 88 bits to reduce memory requirements for hash lookup and hash memory. The hash controller buffers incoming instructions from the loop encoder, aligns them in 1Kbit message blocks and feeds them to the hash engine. The hash engine requires 28 cycles to process a block, thus the hash controller issues a stall signal to the processor in case its buffer is full and the hash engine cannot digest a new message block. Therefore, system calls are handled differently because we observe that they often involve short loops that are executed arbitrarily many times, e.g., string utility functions. Hashing such a short loop path every time it executes, especially for a large number of iterations, would require the hash controller to stall the processor frequently and delibitate performance. Hence, the executed instructions along a loop path are concatenated and stored in plaintext in a dedicated CAM and sent to the hash engine only once when it is first encountered. When the same path is executed again, it is compared with the previously recorded paths in the CAM, and a corresponding counter is incremented when a match is found, without sending it to the hash engine again. The counters are concatenated with the corresponding hash values in the final attestation report.

[4]https://blake2.net/

Upon finalizing a hash computation, the hash controller checks, whether the resulting hash is computed over a path within a loop or not. If it is computed over a path loop, it forwards the resulting hash value from the hash engine synchronized with its corresponding loop_ID to the hash lookup.

D. Hash Lookup

The hash lookup is dedicated to storing and tracking hash values during loop iterations efficiently. Once a hash value is ready, the hash controller forwards it to the hash lookup, which searches within the current loop's list of hash values for a match. If not found, then the hash value is appended to the list. The hash lookup also maintains a counter per loop path which is incremented when its corresponding hash is encountered.

Implementation. To avoid multiple memory accesses due to sequential search of a particular hash value, we implement the hash lookup as a set of CAMs, whose number can be configured based on the system's requirements. A CAM is dedicated for every active loop, so the number of CAMs is determined by the maximum number of nested loops that ATRIUM is configured to track concurrently. Each CAM has a configurable capacity of (n,m) bits, where n is the maximum number of entries and m is number of bits per entry and a counter to maintain the occupied number of entries. When a loop is detected, the hash controller sends the hash lookup to reserve a CAM for it and reset its counter to zero. The CAM holds the computed hash values of a currently executing loop temporarily till the loop exits. Each time a path in the pertinent loop is executed, its computed hash value is looked up in the associated CAM. If a match is not found, i.e., this path has not been executed before, then its hash value is appended to the CAM. When a new loop is detected and all CAMs are occupied, a CAM that was reserved for a loop that already exit (and will not be executed again) is freed and re-used. If a path does not belong to a loop, then its hash value is used to update the hash memory directly.

E. Hash Memory

All computed hash values are stored in a dedicated memory. After the execution of the code region to be attested completes, these hash values are assembled and a digital signature is computed over them. The hash values $H_0\|...\|H_n$ and their signature are then transmitted to $\mathcal{V}rf$.

Implementation. An on-chip hash memory is dedicated to store all computed hash values during a single attestation run of the pertinent code region. The sequence of the storage of the hash values in memory indicates the order of the first occurrence of their corresponding code segments during execution. It is necessary to maintain this order and report $H_0\|...\|H_n$ in the same sequence to $\mathcal{V}rf$ for correctly verifying execution. In our FPGA prototyping of ATRIUM (cf. § VI), we configure the hash memory as on-chip block RAM (BRAM) of configurable capacity with each entry occupying 88 bits for hash digest and 8 bits for its counter. The capacity is configured according to our attestation requirements, i.e., the maximum number of CFG segments an attested code region would consist of. Alternatively, for constrained embedded systems, we can reduce the memory requirements by streaming the hash values (or every batch of them) as soon as they get generated to the $\mathcal{V}rf$.

VI. Evaluation & Security Considerations

A. Performance & Area Evaluation

We implemented ATRIUM in Verilog, interfaced it with the open-source RISC-V Pulpino core [5], and simulated and synthesized it. Performance and functionality were evaluated using a suite of microprocessor benchmarks including *Dhrystone*, *mt-matmul*, *rsort*, *spvm* and *towers*.

Functionality. We extended the Pulpino RTL with ATRIUM and performed cycle-accurate simulation on ModelSim while executing the aforementioned benchmarks. We confirm correct functionality of ATRIUM by comparing simulation results with reference execution profiles of the benchmarks, which we extracted by running the benchmarks on standalone Pulpino without ATRIUM and analyzing the execution trace.

Area and Memory. Area utilization depends on the configurations of the hash lookup and hash memory of ATRIUM. For our evaluation, we configured the hash lookup with 8 CAMs, each CAM with $n = 8$ entries and each entry being $m = 88$ bits. This allows ATRIUM to track up to 8 active nested loops at once with a maximum of 8 different $88-bit$ path hashes per loop. On synthesizing ATRIUM using Xilinx Vivado on a Zedboard (Virtex-7 XC7Z020 FPGA), we show the overall area utilization to be 15% of slice registers and 20% of slice LUTs of this FPGA, while only one 18Kbit BRAM is required for the hash memory.

Performance. Implementation results indicate that ATRIUM can operate at a maximum clock frequency of 70 MHz on a Zedboard (Virtex-7 xc7z020 FPGA) and is, hence, on par with the Pulpino's maximum clock frequency of 50 MHz on the same board. Performance experiments show an overhead of 1.97% for *Dhrystone*, 12.23% for *mt-matmul*, 22.69% for *rsort*, 6.06% for *spvm* and 1.7% for *towers*. Since ATRIUM components run on par with Pulpino, performance loss is caused by the hash function, as the processor stalls occur *only* when the currently executed path has ended and needs to be hashed while the hash engine is still processing the previously executed path and is not ready for input. This overhead is incurred for loops with paths whose number of executed instructions are less than the required number of cycles for the hash engine to finalize its computation (28 cycles for the chosen hash function). To mitigate this overhead, the hash engine should be clocked at a higher frequency than the processor if possible.

B. Security Considerations.

We assume that the used cryptographic primitives are secure. Upon receiving an attestation request, $\mathcal{P}rv$ generates and sends the list of computed hash values $H_0\|...\|H_n$ along with a digital signature computed over it and a nonce provided by $\mathcal{V}rf$ and signed by $\mathcal{P}rv$'s secret key sk. The signature guarantees the authenticity of the attestation report while the nonce ensures its freshness. By verifying the signature, checking the value of the nonce, and comparing the received hashes to their expected values stored in $\mathcal{V}rf$'s database, $\mathcal{V}rf$ gains assurance of the correct execution (both instruction and their control flow) of the current program on $\mathcal{P}rv$. We consider three classes of attacks that can be mounted on ATRIUM.

Malware and Network Attacks. ATRIUM detects malicious software modification introduced by the adversary, as every executed instruction is included in the hash computation. To evade detection, finding a second image that maps to same hash value is required. However, that is infeasible since the hash engine is second pre-image resistant. Forging the signature or replaying an old signature is also not feasible, due to security of signature scheme and to the nonce being long enough.

Runtime Attacks. Since basic block addresses are included in hash computations along with the executed instructions, the hash values computed in ATRIUM reflect the control flow of the executed path. Being tightly integrated with the processor, ATRIUM is guaranteed to track and record every control-flow event executed. An attacker who knows the program code P or CFG(P) can try to bypass ATRIUM by searching for a second pre-image of the corresponding hash. However, by using cryptographically-secure hash function, finding collisions is computationally infeasible.

Physical Attacks. An adversary with physical access to $\mathcal{P}rv$ can try to manipulate the program code in $\mathcal{P}rv$'s memory at runtime, i.e, between time of attestation and time of execution. However, in ATRIUM attestation is performed *during* execution. Therefore, it is guaranteed that *every instruction* that is executed on $\mathcal{P}rv$ will be included in the hash generation, and consequently any manipulation will be detected by $\mathcal{V}rf$, as the generated hash values would not match $\mathcal{V}rf$'s expectations. This defends against TOCTOU attacks that can occur when attestation is *followed* by execution, as was the case for both SMART [9] and C-FLAT [3]. Finally, fault injection attacks which target the SoC clock and cause unintended behavior would also be detected by $\mathcal{V}rf$, as long as the attacks affect the instructions executed or their control flow. Note that, expensive invasive/semi-invasive physical attacks on the SoC are considered out of scope in this work.

VII. Related Work

Attestation Schemes. Existing static attestation schemes such as software-based [14], [20], hardware-based [21], [17], and hybrid [15], [9] attestation schemes are vulnerable to runtime attacks. Control-flow attestation (C-FLAT) aims at enhancing the security of static attestation schemes by additionally hashing the code's execution control flow. This enables the detection of code-reuse and non-control data attacks that divert the execution flow. However, due to frequent hash calculations and context switching (on TrustZone), C-FLAT incurs high performance overhead. LO-FAT [7] leverages hardware assistance to track and measure control flow, thus, overcoming the limitations of C-FLAT and enabling efficient attestation of *uninstrumented* code. LO-FAT, however, incurs significant area overhead due to its on-chip memory requirements (up to 49 36Kbit Block RAMs are used sparsely to store counters of

[5]https://github.com/pulp-platform/pulpino

loops' paths). Finally, in a stronger adversary model with physical access to the prover's device, these schemes are vulnerable to Time of Check Time of Use (TOCTOU) attacks. ATRIUM mitigates this by providing both static and control-flow attestation in a stronger (and more realistic) adversary model efficiently.

Authenticated Memory Modules. Authenticated Memory Modules (such as Intel Authenticated Flash [1]) aim at resisting physical attacks on external memory by preserving the memory's integrity. However, they are insecure under an adversary model with physical access. Moreover, they do not authenticate the control flow of the execution. On the contrary, ATRIUM provides an additional defense against software runtime attacks by coupling the attestation of both the instructions and their control flow with their execution to eliminate any room for TOCTOU attacks.

Memory Authentication. Such schemes [8], [6] aim at resisting physical attacks on external memory. However, they incur high performance overhead by authenticating memory blocks before execution and are susceptible to runtime attacks. ATRIUM detects both runtime attacks and physical attacks on code memory while incurring minimal overhead.

Hardware Security Architectures. Finally, hardware security architectures (such as Intel SGX) provide memory authentication as well as static attestation. However, such architectures are not designed to target low-end embedded devices. Furthermore, they only provide static attestation and therefore cannot meet the goals that we target. Nevertheless, they provide security features complementary to our work.

VIII. Conclusion

Due to the ubiquity of interconnected embedded systems, software running on these devices have become vulnerable to remote software attacks. Previous attestation schemes have been proposed to detect these attacks while always ruling out physical attacks. In this paper, we showed that physical attacks on the system's code memory are indeed feasible. We presented a hardware-based efficient scheme ATRIUM that allows precise attestation of both executed instructions as well as their control flow. ATRIUM is the first attestation scheme to provide security guarantees against a stronger adversary with physical access to code memory, and does not require any code instrumentation (compliant to legacy software) or instruction set extension. Our proof-of-concept implementation is highly efficient with reasonable performance impact on the attested software at an expense of minimal area overhead and memory.

Acknowledgments. We thank the authors of SMART for sharing their scheme. This work was supported by the German Science Foundation CRC 1119 CROSSING, the German Federal Ministry of Education and Research (BMBF) within CRISP, the EU's Horizon 2020 research and innovation program under grant No. 643964 (SUPERCLOUD), the U.S. Department of Energy through the Early Career Award (DE-SC0016180), the National Science Foundation Graduate Research Fellowship Program under grant No. 1144246, and the Intel Collaborative Research Institute for Secure Computing (ICRI-SC).

References

[1] Intel Authenticated Flash. www.design-reuse.com/articles/16975.
[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. "Control-flow integrity: Principles, implementations, and applications". *ACM Transactions on Information and System Security*, 2009.
[3] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. "C-FLAT: Control-flow attestation for embedded systems software". In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.
[4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. "When good instructions go bad: Generalizing return-oriented programming to RISC". In *ACM SIGSAC Conference on Computer and Communications Security*, 2008.
[5] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. "Non-control-data attacks are realistic threats". In *USENIX Security Symposium*, 2005.
[6] R. de Clercq, R. De Keulenaer, P. Maena, B. Preneel, B. De Sutter, and I. Verbauwhede. "SCM: Secure code memory architecture". In *ACM Symposium on Information, Computer and Communications Security*. ACM, 2017.
[7] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi. "LO-FAT: Low-overhead control flow attestation in hardware". In *Design Automation Conference*, 2017.
[8] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres. "Hardware mechanisms for memory authentication: A survey of existing techniques and engines". In *Transactions on Computational Science IV*. 2009.
[9] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. "SMART: Secure and minimal architecture for (establishing dynamic) root of trust". In *Annual Network and Distributed System Security Symposium*, 2012.
[10] A. Francillon and C. Castelluccia. "Code injection attacks on Harvard-architecture devices". In *ACM SIGSAC Conference on Computer and Communications Security*, 2008.
[11] V. Haldar, D. Chandra, and M. Franz. "Semantic remote attestation: A virtual machine directed approach to trusted computing". In *Virtual Machine Research And Technology Symposium*, 2004.
[12] Hewlett-Packard. "Data execution prevention", 2006.
[13] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. "Data-oriented programming: On the effectiveness of non-control data attacks". In *IEEE Symposium on Security and Privacy*, 2016.
[14] R. Kennell and L. H. Jamieson. "Establishing the genuinity of remote computer systems". In *USENIX Security Symposium*, 2003.
[15] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. "TrustLite: A security architecture for tiny embedded devices". In *ACM SIGOPS EuroSys*, 2014.
[16] T. Kornau. "Return oriented programming for the ARM architecture". Master's thesis, Ruhr-University Bochum, 2009.
[17] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. "New results for timing-based attestation". In *IEEE Symposium on Security and Privacy*, 2012.
[18] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. "Code-pointer integrity". In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.
[19] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. "SoK: Automated software diversity". In *IEEE Symposium on Security and Privacy*, 2014.
[20] Y. Li, J. M. McCune, and A. Perrig. "VIPER: Verifying the integrity of peripherals' firmware". In *ACM SIGSAC Conference on Computer and Communications Security*, 2011.
[21] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. "Copilot – A coprocessor-based kernel runtime integrity monitor". In *USENIX Security Symposium*, 2004.
[22] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. "Design and implementation of a tcg-based integrity measurement architecture". In *USENIX Security Symposium*, 2004.
[23] H. Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In *ACM SIGSAC Conference on Computer and Communications Security*, 2007.
[24] L. Szekeres, M. Payer, T. Wei, and D. Song. "SoK: Eternal war in memory". In *IEEE Symposium on Security and Privacy*, 2013.