Author's Preprint: For Educational Use Only
To Appear at The 45th International Symposium on Computer Architecture - ISCA 2018, LA, USA, June 4 - 6th

Practical Memory Safety with REST

Kanad Sinha Simha Sethumadhavan

Department of Computer Science
Columbia University
New York, NY, USA
{kanad,simha}@cs.columbia.edu

***Abstract*—In this paper, we propose Random Embedded Secret Tokens (*REST*), a simple hardware primitive to provide content-based checks, and show how it can be used to mitigate common types of spatial and temporal memory errors at very low cost. *REST* is simply a very large random value that is embedded into programs. To provide memory safety, *REST* is used to bookend data structures during allocation. If the hardware accesses a *REST* value during execution, due to programming errors or adversarial actions, it reports a privileged memory safety exception.**

Implementing *REST* requires 1 bit of metadata per L1 data cache line and a comparator to check for *REST* tokens during a cache fill. The software infrastructure to provide memory safety with *REST* reuses a production-quality memory error detection tool, AddressSanitizer, by changing less than 1.5K lines of code.

***REST* based memory safety offers several advantages compared to extant methods: (1) it does not require significant redesign of hardware or software, (2) the overhead of heap and stack safety is 2% compared to 40% for AddressSanitizer, (3) the security of the memory safety implementation is improved compared AddressSanitizer, and (4) *REST* based memory safety can mitigate heap safety errors in legacy binaries without recompilation or source code. These advantages provide a significant step towards continuous runtime memory safety monitoring and mitigation for legacy and new binaries.**

***Keywords*-memory safety, hardware support, REST, Random Embedded Secret Tokens, AddressSanitizer, privileged memory safety exception, microarchitecture, load store queue, cache microarchitecture.**

I. Introduction

Memory corruption errors have been one of the most persistent and long-standing problems in computer security. However, practical and effective solutions to this challenge, although critical to secure program operation, remains an elusive goal to this day. In fact, heap-based memory attacks, exploiting out-of-bounds heap read/writes and use-after-free (UAF) bugs alone, accounted for 80% of root causes that led to remote code execution (RCE) in Microsoft software in 2015 [1].

Previous hardware techniques to address memory safety concerns are broadly based on two approaches — whitelisting safe memory regions and blacklisting (some portion of) unsafe memory regions. Previous work in the former approach, broadly referred to as bounds checking, associates metadata with every pointer indicating the bounds of the data structure it can legitimately access, and flagging any access outside those bounds as memory errors. In the latter approach, commonly called the tripwire approach, critical locations in the address space (for instance, both ends of an array) are marked invalid and any access to them raises a memory violation exception.

Whitelisting approaches [2], [3], [4], [5], [6], [7] offer stronger security guarantees since they monitor all memory accesses against exact bounds. Another advantage to per-pointer metadata is that some of these mechanisms also maintain liveness/version information about data structures they point to, thus detecting dangling pointers in addition to out-of-bound errors. However, they suffer from one or more of the following problems.

① **Performance Overhead.** Since they monitor every pointer dereference, the performance overhead scales with the number of dynamic pointer references. For each of these references there is at least one additional memory instruction for loading the meta data and one comparison operation for checking the data. Even if some overhead can be mitigated by optimizations such as caching, the energy overheads due to the additional instructions are not easily mitigated.

② **Implementation Overhead.** They usually require significant hardware modifications including modifications to the cache hierarchy [2], [4], execution pipeline [2], [4], [7], or even addition of coprocessors [6].

③ **Inaccurate/incomplete Coverage.** Since most of them rely on static pointer analyses for metadata propagation during pointer operations, any inaccuracy in pointer identification leads to incorrect/unstable program behavior. This is especially problematic in the C-memory model, which allows interchangeability between pointer and native data types [8]. Additionally, this also necessitates source code availability, thus preventing such techniques from being compatible with legacy binaries.

Tripwires, originally proposed for software, are not a commonly explored technique in hardware [9], [10]. These techniques provide a relatively fast mechanism for marking memory locations invalid. By associating metadata with the locations instead of their pointers, they avoid metadata propagation costs, thus mitigating some drawbacks of whitelisting techniques. However, this comes at the expense of weaker security guarantees since they do not detect all spatial violations (specifically ones that access unmarked regions). In fact, these techniques target a specific access pattern which is commonly responsible for memory overflows. This pattern manifests itself when the program sequentially starts accessing locations beyond the bounds of the data structure (in a loop, for instance). Previous attempts at hardware support for tripwire implementation have required non-trivial hardware modifica-

tions (including storage of metadata) and/or incurred non-trivial performance penalty. Furthermore, previous hardware techniques in this category only focus on detecting out-of-bounds accesses and do not address temporal memory safety even though it accounted for 51% of RCE exploits in Microsoft software in 2015, whereas the former accounted for 28.5% [1].

Additionally, checks performed by previous schemes were *tag-based*, in that they use metadata tags, stored in a region separate from program data, to compare and verify access validity. This, in turn, requires (explicit or implicit) out-of-band fetching and processing of metadata.

In this paper, we propose *Random Embedded Security Tokens* (*REST*), a hardware primitive for content-based checks, and describe a framework based on a primitive enabling programs to blacklist memory regions at a low overhead. This primitive allows the program to store a long unique value, a *token*, in the memory locations to be blacklisted and issues a privileged *REST* exception if it is ever touched with a regular access. We propose a low overhead, low complexity microarchitecture for detecting these tokens. When an L1 data cache line is filled, that memory line is checked for the *REST* token value and if so, marked as such. If a memory instruction accesses that marked line, we throw an exception. These hardware modifications are trivial, requiring no modifications to either the core design, or the coherence and consistency implementations of the cache, even for multicore, out-of-order processors. Ours is also the first scheme to rely on *content-based* checks wherein the metadata is stored alongside program data and requires no modification of the program's overall memory layout. Token checks are performed directly on all data accessed by the program and requires no behind-the-scene metadata processing.

The rest of our framework is based on a software tripwire-based scheme, AddressSanitizer (ASan) [11], which consists of a compilation framework and runtime library that automatically fortifies programs against memory errors without any programmer effort. ASan is a highly popular memory error detector, used in the testing infrastructure of production softwares such as Firefox [12] and Chromium [13]. However, due to its high performance overhead (~1.4x), it is mainly used for software testing and debugging, not in deployment builds. Comparatively, *REST* incurs an overhead of 2% on the SPEC benchmarks while not only providing the same scope of protection as ASan, but even improving its security in several aspects. Moreover, our technique is also able to provide heap safety for legacy binaries at similar overheads. Additionally, as we show later, the observed overheads are completely attributable to the software framework; our hardware primitive incurs nearly zero additional performance overhead, and has negligible implementation complexity.

We illustrate the basic idea of our defense with a simplified version of CVE-2014-0160 [14], a bug commonly known as the Heartbleed vulnerability reported in OpenSSL 1.0.1, as shown in the code shown in Listing 1.

Line 7 in the listed routine contains the overflow bug wherein the payload length, `payload`, is used to determine the size of data to be copied into the response packet without checking its validity. The resulting exploit can then be used

```
1  int tls1_process_heartbeat(SSL *s) {
2    unsigned char *p = &s->s3->rrec.data[0];
3    unsigned short hbtype = *p++;
4    unsigned int payload;
5
6    /* Attacker-controlled memcpy length */
7    n2s(p, payload);
8
9    if (hbtype == TLS1_HB_REQUEST) {
10     unsigned char *buffer =
11                   OPENSSL_malloc(payload);
12
13     /* Vulnerable OOB memory read */
14     memcpy(buffer, p, payload);
15     ...
```

Listing 1: Heartbleed out-of-bounds memory read bug.

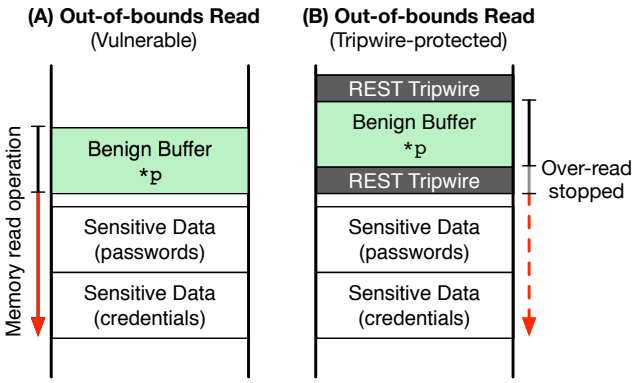

Fig. 1: (A) Unsanitized `memcpy` bug reads sensitive data outside the benign buffer. (B) *REST* tokens placed around the buffer detects this out-of-bounds access.

to leak sensitive information such as passwords, usernames, secret keys *etc.*, to the client. Furthermore, common protections involving (stack or heap) canaries would be unable to detect this attack, since it involves a read overflow and does not otherwise corrupt any program state. To prevent this, *REST* tokens are placed around the source buffer to be copied, so that when access goes beyond its bounds, a security exception is triggered, as shown in Figure 1.

II. Motivation

Functionally, *REST* provides similar safety features as ASan, a state-of-the-art memory error detector widely used for verification and debugging. Despite its effectiveness, it is not used as a live security scheme due to its performance overheads.

ASan implements a software tripwire-based system, wherein blacklisted zones (also called *redzones*) are placed around sensitive data structures. It then detects erroneous program behavior that leads to illegitimate accesses of these location (in case of an overflow, for instance). To do so, ASan primarily relies on two techniques — shadow memory and memory access instrumentation (see Figure 2). Firstly, it reserves a chunk of memory, called *shadow memory*, that contains metadata and should never be explicitly accessed by the program. The rest of

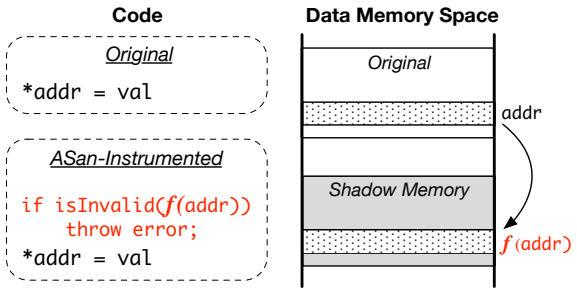

Fig. 2: Code and address space transformation done by ASan. Memory accesses are instrumented to check against the corresponding value in the shadow memory (dark region in figure), calculable with a simple mapping function, f.

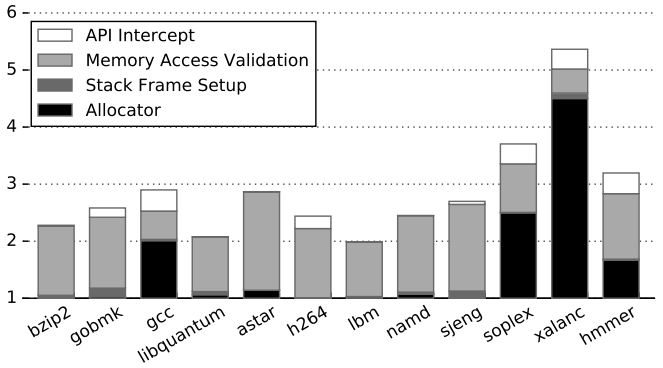

Fig. 3: Breakdown of various sources of overhead in ASan with respect to a plain binary using `libc`'s allocator.

the address space maps to its corresponding shadow location via a simple mapping function. Additionally, ASan imposes memory-safe program behavior by checking the validity of every memory access against the metadata for the accessed location. This is achieved by statically instrumenting the program to insert checks before every memory access. When data structures are deallocated, the corresponding regions are marked invalid by zeroing out the corresponding metadata.

Sources of Overhead. In terms of performance, ASan has four major sources of overhead. ① ASan uses a *custom allocator* designed with security in mind that maintains separate pools for free memory (from which new allocations are made) and deallocated memory (consisting of recent deallocations), and allows virtually no allocation reuse in order to prevent use-after-free (UAF) errors. Hence, it is slower than other allocators which are primarily designed with performance as a first-order feature. ② ASan inserts code at function prologues and epilogues to *modify the stack frame* by inserting and aligning stack variables in order to deter stack attacks. ③ *Instrumentation* for validating memory accesses, as discussed above, also contributes towards ASan's slowdown. ④ Furthermore, since memory checks cannot be inserted in third party libraries, ASan partially mitigates the problem by *intercepting common* `libc` *data-handling API calls* (e.g.,`strcpy` and `memcpy`) to verify that no invalid access occurs therein for the particular set of arguments.

Figure 3 provides a breakdown of these components for the SPEC CPU2006 benchmarks simulated on an in-order core[1]. As we see in the figure, memory access checks (③ and ④) account for the most persistent and grievous source of overhead, although the allocator also contributes significantly for benchmarks that make frequent heap allocations. In the subsequent sections, we show how our scheme removes the overheads associated with most of these components.

Notably, ASan's developers also consider potential hardware assistance [15] to speed up metadata lookup and memory access checks transparently by encoding the corresponding logic within a single architectural instruction in a design similar to Watchdoglite [5]. As such, ASan-fortified programs could compress the entire memory-access validation into a single instruction, thus optimizing the expensive operations, but not necessarily removing them. Furthermore, although Watchdoglite has been shown to be highly effective for memory safety in its own respect, such a design would suffer from some of the drawbacks of bounds checking schemes discussed earlier and would necessarily require recompilation. We discuss and contrast similar hardware techniques in more detail in §VII.

[1]The memory side configuration is same as in Table II.

III. Hardware Design

Since *REST* hardware aims to detect and flag accesses to tokens, our main challenge is to be performant by hiding latencies associated with additional memory checks, while maintaining existing microarchitectural optimizations and ensuring the integrity of token semantics. Modifications for *REST* consists of extending the ISA with two new instructions and an exception type, as well as microarchitectural modifications to support them with minimal overhead. We discuss these aspects of the *REST* primitive design below.

A. ISA Modifications

The width of the token is that of a cache line (64B in our system), and its value is held in a token configuration register (which is not directly accessible to user-level applications). Two instructions are added to set (store) and unset (remove) tokens in the application:

① `arm <reg>` This instruction stores a token at location specified in register `reg`, which should be capable of addressing the entire address space. The implicit operand in this instruction is the token value stored in the token configuration register. The specified location has to be aligned to the token width, otherwise a precise invalid *REST* instruction exception is generated.

② `disarm <reg>` This instruction overwrites a token at location specified in the register `<reg>`, which should be capable of addressing the entire address space, with the value zero. The specified location also has to be aligned to the token width, otherwise a precise invalid *REST* instruction exception is generated. Additionally, in case there is no token at the location, a *REST* exception is generated as well.

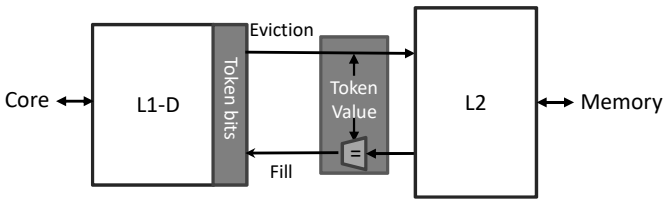

Fig. 4: Hardware modifications for *REST* include an extra metadata bit per cache line in L1 data cache indicating whether it contains a token, and the token detector to examine incoming data from lower caches and fill the token value into evicted lines.

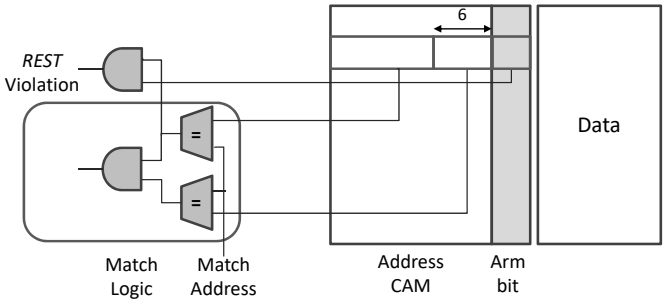

Fig. 5: Modifications to the LSQ. Added structures are noted in darker shade.

When a *REST* exception is triggered, the exception is handled by the next higher privilege level. If the exception is generated at the highest privilege mode, we consider it a fatal exception. We also assume the faulting address is passed in an existing register.

Setting the token value is done through a store instruction that writes to a memory-mapped address. Depending on the token width, one or more stores might be necessary to set the full token value. This operation can only be performed by a higher privileged mode.

We also provide two modes of operation, *debug* and *secure*. The secure mode is expected to be the typical mode of operation for programs in deployment and does not guarantee precise recovery of program state on a *REST* exception (behavior for other exceptions remains unchanged). In the debug mode, the entire program state at the time of *REST* exception can be precisely recovered by the exception handler. Thus, this mode is intended for use by developers. The current mode of operation can be configured by setting a bit in the token configuration register.

B. Microarchitecture

In our design, loads and stores check the accessed data against the token value and raise an exception in case of a match. Thus, logically each load becomes a load followed by a comparison of the loaded value with the token, while a store becomes a load of the value to be overwritten, a comparison with the token value, followed by the store. Additionally, reading and/or writing a 64B token value would involve data transfers over multiple cycles, since data buses are narrower. Naively implemented, this could increase the latency and energy of memory operations significantly.

We show a novel construction for *REST* that minimizes changes to load store pipelines and latency for memory operations. Our key observation is that checks necessary for the *REST* system can be performed when the cache lines are installed or accessed instead of explicitly fetching the values and checking them.

Cache Modifications. We extend each cache line in the L1 data cache to include one additional bit to indicate if that line contains a token. Note that since tokens are aligned, a token is guaranteed to be contained within a single line. When a cache line is being installed, the value of that line is compared to the token value register and in case of a match, the token bit corresponding to that line is set. Since cache fills typically happen over multiple cycles the token comparison can be decomposed into small manageable compare operation, say a 32b compare per cache fill stage, to reduce energy. After the fill, memory operations that access lines with the token bit set are flagged to throw a *REST* exception.

A disarm instruction unsets the token bit corresponding to the accessed line and concurrently zeroes out the entire cache line. Since such an operation involves all data banks of the cache, disarm writes incur an additional, typically one cycle, latency. Additionally, disarms raise a *REST* exception if the token bit is not set on the destination line, thus ensuring that the program can only disarm armed locations. The arm instruction sets the token bit of the accessed line, but does not write the token value into it; the token values are written out when the line is evicted from the L1 data cache. This construction ensures that arm operations that hit in the cache complete in a single cycle, despite being a wide write. Our construction works naturally for write-allocate caches, which is one of the most commonly used allocation policies supported in current microarchitectures.

LSQ Modification. Since arm and disarm instructions write values, they are functionally stores and handled as such in the microarchitecture with one key difference. Unlike stores, the arm and disarm instructions should not forward their values to younger loads, as this will violate the invariant that the *REST* token must be a secret. One simple way to provide this invariant is to serialize the execution of arm and disarm execution, i.e., ensure that an arm or disarm instruction is the only inflight instruction when it is encountered in the decode stage. This option, while simple to implement, can introduce significant performance penalities.

Instead of serialization, we next describe design to prevent such forwarding in a common (and complex) structure used to support store to load forwarding, the load-store queue (LSQ). Consider a scenario where an arm request is closely followed by a read to the same cache line. In this case the load may “hit" the in-flight arm in the LSQ, thus forwarding an otherwise illegal read. When this case is encountered, we throw a privileged *REST* exception.

This exception support can be implemented without any additional state or impact on LSQ access timing. To do so,

Action	LSQ	Cache Hit	Cache Miss
Arm	Create entry in SQ, tag as arm.	Set token bit	Fetch line, set token bit.
Disarm	Raise exception if SQ has disarm for same location. Else insert entry with no store value in SQ, tag as disarm.	If token bit unset, raise exception. Else clear line, unset token bit(s).	Fetch line, set token bit if it has token. Proceed as hit.
Load	If value can be forwarded from armed SQ entry, raise exception. As usual otherwise.	If token bit set, raise exception. Else read data.	Fetch line, set token bit if it has token. Proceed as hit.
Store (Secure)	Raise exception if SQ has arm for same location. As usual otherwise.	If token bit set, raise exception. Else write data.	Fetch line, set token bit if it has token. Proceed as hit.
Store (Debug)	Raise exception if SQ has arm for same location. As usual otherwise.	If token bit set, raise exception. Else write data.	Fetch line, set token bit if it has token. Delay store commit till ack from L1-D.
Coherence Msgs.	N/A	As usual.	As usual.
Eviction	N/A	If token bit set, fill token value in outgoing packet.	N/A

TABLE I: Actions taken on various operations for L1-D cache hits and misses.

we incorporate the *REST* violation check into the existing matching logic simply by breaking the match down to perform two matches — one an address match for the cache line address and another for the remaining — and adding a few logic gates (as shown in Figure 5). Additionally since the arm and disarm write values are implicit and known by the cache, we do not attach a value with the corresponding entry in the store queue. With these modifications, LSQ access latencies and data widths remain unchanged despite the introduction of very wide writes. Such address modifications may be necessary at other places in the microarchitecture where store to load forwarding may occur.

Exception Reporting. We can further optimize the performance cost of *REST* by being flexble about how and when exceptions are reported. Supporting precise exceptions with *REST* requires disabling performance optimizations such as critical-word first, and early and eager commit of stores that are common in modern processors. However, *REST* exceptions do not have to reported precisely especially when it is used for monitoring for security violations during deployment as in these cases the user is typically interested in knowing if a security violation occurred or not, and not the state of the machine when the violation occurred.

If the L1 data cache supports critical-word first fetching, the access request may be satisfied before the whole line has arrived and a match determined. This creates the possibility of a delay between load commit and the security check, especially when the load is at the head of the ROB and is committed as soon as the critical word arrives but the entire line has not. In the debug mode, loads are not released from the MSHRs as long as the delivered word partially matches the token value. On a mismatch, the load is released without any performance penalty. In the secure mode, *REST* exception is reported independent of the load commit.

Additionally, since stores are committed from the ROB as soon as the store/arm/disarm becomes the oldest instruction, *REST* violations due to a faulty access might not be resolved in time. By the time the violation is detected at the cache and the response is received at the ROB, the offending instruction may have retired. This will result in an imprecise *REST* exception. In the debug mode, we guarantee precise exceptions by delaying store commit until writes completion.

Modifying Token Width. The token width can be reduced for security and performance reasons. For instance, instead of a full cache line width, half or quarter cache line tokens may be used. Most changes described above can be simply scaled to accommodate this. For instance, the token value register can be smaller, and the number of token bits per line will increase to 2 and 4 for 32- and 16-byte tokens respectively.

IV. Software Design

The *REST* primitive described above provides programs the capability to blacklist certain memory locations and disallow regular accesses to them. In this section, we describe how programs can leverage this primitive to obtain spatial and temporal memory safety with little to no changes in its construction and/or layout.

A. Userlevel Support

We base our software design on ASan, which is a highly popular open-source memory error detection tool. *REST*'s software framework, however, uses tokens instead of metadata to denote redzones. This obviates two major components of ASan's original design. Since our hardware continuously detects access to tokens without software intervention, monitoring every program read and write in software becomes unnecessary. Thus, memory operations no longer need to be instrumented for checking access validity. Secondly, since *REST* tokens do not require separate maintenance of metadata, the need for shadow memory is eliminated as well. Combined, this essentially eliminates the two major sources of ASan's performance and memory overheads, simplifying its implementation complexity.

Protecting the Stack. As shown in Figure 6, protecting vulnerable stack variables involves placing redzones around it. This is done by code added at the function prologue, so redzones isolate these variables from the other local variables. The size of each redzone is chosen as a multiple of the token width and is based on the size of the data structure. Subsequently, overflows during the frame's lifetime are detected

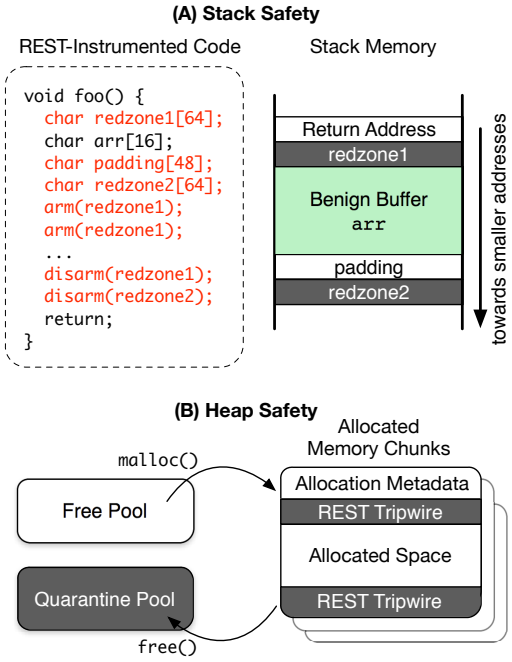

Fig. 6: (A) For stack safety we instrument the program to insert tokens around vulnerable buffers. (B) Our allocator provides heap safety by surrounding allocations with tokens and blacklisting deallocated regions in the quarantine pool.

when accesses go past their boundaries and into one of the redzones. Code is also inserted at the function epilogue to clean up the tokens so that future frames inherit a clean stack.

Since the above changes involve modifying the stack layout, *REST* requires that binaries be compiled with our plugin. However, since stack attacks have become an insignificant threat vector in recent years [1], users may also choose to forego stack protection, if performance is a concern, and just opt for heap protection as described next.

Protecting the Heap. *REST* secures the heap with a custom allocator adapted from ASan. Spatial heap protection is provided by ensuring that the allocator surrounds every allocation with redzones (see Figure 6). These redzones not only separate the allocations from each other but also from the metadata.

Temporal bugs are prevented by filling all freed allocations with tokens and placing them in a separate *quarantine pool*, instead of the pool of free memory from which new allocations are assigned. They remain there until the free memory pool has been sufficiently consumed at which point, they are disarmed and released for reallocation. Hence, UAF attacks are mitigated since freed allocations remain blacklisted and any attempts at accessing them via dangling pointers or double frees are caught.

We make one modification to ASan's free pool management however. ASan originally maintains the invariant that all entries in its free and quarantine pool be blacklisted. This necessitates blacklisting newly mapped region from the system, and mark them valid just before allocation. For *REST* we relax the invariant to guarantee that only quarantined regions are blacklisted while those in the free pool are zeroed. This is because blacklisting, in our case, involves storing tokens all over the newly mapped regions and is hence slower than just rewriting corresponding metadata as is done by ASan. Our invariant is maintained for reused regions since disarms zero out memory before they are moved to the free pool and reallocated, thus avoiding uninitialized data leaks.

One key advantage of our protection mechanism is that it works with legacy binaries. Since *REST* performs memory access checks in hardware, heap protection in our case does not require any instrumentation of the original program and can thus be availed even by legacy binaries, as long as our custom allocator is used (with `LD_PRELOAD` environment variable in Unix-based systems, for instance).

B. System Level Support

At the system level, we propose having a single token value. As will be discussed in §V, the token widths are sufficiently long that the chances of a random program value matching a token is vanishingly small (see §V-B). However, leaking this value via physical or side-channel attacks might still be possible and would compromise the entire system. So periodically this token value can be rotated (at reboot, for instance). Our design for heap safety allows this model without the need for recompilation.

Alternatively, a unique token value could be used for every process with the OS maintaining them across context switches. This design requires some changes to the OS such as the generation of token values and the ability to deal with tokens from different processes when processes are cloned or communicate with each other.

V. Hardware/Software Security

A. Threat Model

In line with recent related work regarding memory error based attacks and defenses, we assume the following in and of our system. The target program has one or more memory vulnerabilities, that can be exploited by an attacker operating at the same privilege level to gain arbitrary read and/or write capabilities within the execution context. We do not make any assumptions as to how these vulnerabilities arise or what attack vectors are used to exploit them. We also assume that the target has common hardware defenses available in most systems today (e.g., NX-bit). Furthermore, we assume that the hardware is trusted and does not contain and/or is not subjected to bugs arising from improper usage parameters resulting in glitching, physical, or side-channel attacks.

B. Hardware Discussion

In this section, we discuss the security implications of our token primitive independent of the software framework utilizing it.

• **Token Width.** A key assumption of our design is that token detection does not suffer from false positives, which occur when token exceptions are triggered by a legitimate chunk of program data. Three conditions have to be met for this.

① The data chunk equals token value,

② It is aligned to token width, and
③ It is fetched into the L1 data cache, thus passing through the token detector. If data transiently acquires the token value while already in L1 data cache or any other part of the memory subsystem, no exception is raised.

To avoid false positives, it is therefore critical not only to choose a properly random token value but also an appropriate token width. In our design we choose a width of 512 bits, which makes the chances for a program data chunk causing a false positive less than $\frac{1}{2^{512}}$[2]. If this degree of overprovisioning is considered excessive, smaller token widths of 256 bits or even 128 bits could be used. As discussed in §III, these values should entail minimal changes in our original design and can even be supported simultaneously.

• **Immutability and Unmaskability.** *REST* makes sure that once a token is set, it can only be removed through a disarm operation and cannot be otherwise overwritten (or even read) by any process at the current privilege level. Additionally, *REST* exceptions cannot be masked from the same privilege level. These measures ensure that adversaries cannot exploit inter-process, inter-core, or inter-cache interactions to bypass token semantics.

• **Detector Placement.** We place our detector at the the L1 data cache in order to keep the other caches unmodified and hence, minimize design costs. Consequently, however, *REST* does not catch token accesses via means that completely sidestep the cache (e.g., DMA).

C. Software Discussion

While *REST* is based on ASan, it improves upon ASan's security in a number of ways. In this section, we elaborate upon the weaknesses of ASan, if/how *REST* mitigates them, and whether we introduce any vulnerabilities of our own.

• **False Negatives.** Token width affects token alignment and therefore, the target data structures[3]. Imposing this granularity on program data, in turn, introduces small gaps between variables. For instance, in Figure 6, *REST* adds a pad space adjacent to an array to conform to the granularity requirement (64B in the figure). This introduces the scope for false negatives, wherein *REST* is unable to detect overflows that are small enough to spill into the pad, but not into the token itself. This implies that although we still protect against read/write overflows, our system is vulnerable uninitialized data leaks in the stack [17], which can be simply prevented by zeroing out the padding or using narrower tokens. Uninitialized data leaks are not a problem in the heap, however, due to our invariant that all regions in our allocator's free pool are zeroed.

• **Brute-force Disarm.** Our decision to mandate precise specification of an armed location while disarming is to counter a scenario when an attacker has somehow obtained control of a disarm gadget (i.e., can influence its address argument), but does not accurately know the layout regarding which memory locations are specifically armed. In such a scenario, this design decision prevents attackers from blindly disarming swathes of memory regions. Properly compiled code, however, should have no problems due to this stipulation.

• **Privilege.** Although used in some security mechanisms [18], ASan was primarily developed for debugging. While it can serve as a security tool under weak threat models and performance requirements, realistically it has limited utility as one. This is primarily because its framework is implemented at the same privilege level as the program itself. While the location of shadow memory is randomized, it remains open to memory disclosure attacks, upon which the metadata can be easily tampered with. Memory access monitoring, while statically baked into the program, can also be subverted with carefully crafted code gadgets or even simple code injection. We overcome this issue by raising a *REST* violation on a token, regardless of privilege.

• **Handling `setjmp/longjmp`.** Since the program can neither probe for the presence of a token, nor does it keep a log of all armed locations, disarming necessarily needs to be carried out in the presence of a known reference point. For the stack, frames serve this purpose, i.e., for a given function, arms/disarms occur at fixed offsets within the frame. Consequently, we could not extend *REST*'s protection to support programs that use `setjmp/longjmp` since these instructions alter the stack layout. ASan takes a very conservative approach in such cases by zeroing out the metadata, and hence whitelisting the entire region of the current stack. We cannot take the same approach since we do not keep track of active tokens on the stack. Providing a secure and cheap mechanism for handling this case remains a topic of future research.

• **Predictability.** Our design, as well as ASan's, suffers from predictable layout as attackers can simply jump over redzones (countered to some extent by adjusting redzone size according to the buffer size). Although we do not use it in our system, we recommend that *REST* be used in conjunction with some variant of layout randomization, depending on the usage scenario. Layout randomization for the heap [19], [20] and stack [21], [22] has already seen a significant amount of work in recent times and has been shown to be easily and effectively applicable. Alternatively, programs could also sprinkle arbitrary tokens across the data region in a configurable manner to catch such attempts.

• **Temporal Protection.** In terms of temporal safety, ASan's, and consequently our guarantees are incomplete since we unmark previously allocated blocks when we reallocate them, after which point, dangling pointers or double frees can no longer be detected. This can be prevented to some extent by using heuristics such as reducing reallocation predictability by maintaining some degree of randomness for new allocations and ensuring that its entropy is never compromised by maintaining a large enough free memory pool. In our setup, however, we rely on ASan's existing allocation algorithm and do not augment it any further.

• **Composability and Coverage.** In order for ASan to be effective, *all* memory accesses to user data need to be monitored. Hence, it is essential that all software modules (the

[2]For simple reference, a maximum of 2^{48} token-aligned data chunks can reside in a 64b address space simultaneously. Additionally, a modern system operating at 3GHz would need ~10^{145} years to guess a 512b random value via simple increment operations.

[3]ASan also imposes alignment on protected data structures [16].

Core	Frequency	2 GHz
	BPred	L-TAGE, 1+12 components, 31k entries total
	Fetch	8 wide, 64-entry IQ
	Issue	8 wide, 192-entry ROB
	Writeback	8 wide, 32-entry LQ, 32-entry SQ
Memory	L1-I	64kB, 8-way, 2 cycles, 64B blocks, LRU replacement, 4 20-entry MSHRs, no prefetch
	L1-D	64kB, 8-way, 2 cycles, 64B blocks, LRU replacement, 8-entry write buffer, 4 20-entry MSHRs, no prefetch
	L2	2MB, 16-way, 20 cycles, 64B blocks, LRU replacement, 8-entry write buffer, 20 12-entry MSHRs, no prefetch
	Memory	DDR3, 800 MHz, 8GB, 13.75ns CAS latency and row precharge, 35ns RAS latency

TABLE II: Simulation base hardware configuration.

main program and shared libraries) be compiled with ASan support. Consider a situation where the program itself has been compiled as desired but a third-party library has not. In such a case, if the library has faulty code resulting in buffer overflow and it operates on a ASan-augmented buffer, the scope for exploitation still remains since read/writes in the library are not being monitored. The reverse situation also applies when the fortified code is in the ASan-augmented program but the data originates in the library, since the foreign buffer does not have the right bookends. Hence, ASan requires both access monitoring and metadata maintenance, one or both of which might break when using non-ASan augmented modules. Analysing and instrumenting the shared libraries at runtime would incur a huge performance penalty (as demonstrated by tools like Valgrind [23])[4].

REST relaxes this requirement greatly by not requiring explicit access monitoring. Thus, as long as the data itself is properly bookended, it does not matter whether the code accessing it has been instrumented or not. As such, it is more compatible with untreated external libraries. Since token access also generates exceptions at higher privileged levels, token manipulation via syscalls is also prevented.

VI. Evaluation

A. Experimental Setup

We implement *REST* in the out-of-order CPU model of gem5 [24] for the x86 architecture. Due to its limited support for large memory mappings, we were unable to run x86/64 binaries since gem5 could not accommodate ASan's shadow memory requirements. Consequently, we simulate 32-bit i386 binaries of the SPEC CPU2006 C/C++ benchmark on the modified simulator in the syscall emulation mode with a configuration shown in Table II. The `arm` and `disarm` instructions were implemented by appropriating the encodings for x86's `xsave` and `xrstor` instructions respectively, which are themselves unimplemented in gem5.

The benchmarks were compiled with Clang version 5.0.0 with `"-O3 -mno-omit-leaf-frame-pointer -mno-sse -fno-optimize-sibling-calls -fno-omit-frame-pointer"` flags. We run these programs to completion with the test input set. Since executions with these inputs spend a significant amount of time initializing (and allocating) compared to the ref input set, this choice of input sets should reflect on our results adversely since the overheads associated with our allocator will not be amortized with computation as well as in the case of ref inputs.

[4]ASan mitigates this to some extent by intercepting common library calls (like `strcpy`), checking the input data appropriately before the call.

B. Overheads

To evaluate *REST*, we compare it against two baselines — unsafe, plain binaries using the stock `libc` allocator, and binaries fortified with ASan. We evaluate two modes, secure with imprecise exception and debug with precise exceptions, for two defensive scopes, full (i.e., stack and heap) and heap only. Additionally, we present another category of numbers for perfect, zero overhead *REST* hardware (referred as PerfectHw) as a limit study of the current hardware design's optimality. The results are presented for each benchmark in Figure 7 as slowdowns relative to the unsafe binary. In addition, we show the weighted average mean overhead[5] as well (referred as the WtdAriMean). For reference, the geometric mean of the overheads[6] is also presented in the figure, but for the following discussion, the cited values refer to the weighted average, not geometric mean[7].

***REST* vs. Baseline.** In the secure mode, *REST* shows an overhead of 2% while providing full or heap safety respectively. For the debug mode, the corresponding values are 25% and 23% respectively. In both modes, we find that the overall trend is roughly consistent with the results presented in Figure 3. Relative to ASan, *REST* does not perform memory checks (via explicit program instrumentation or `libc` call interception). In case of just heap safety, it additionally does not bear the cost of stack instrumentation. Accordingly, we observe that the numbers for *REST*'s full safety follow the expected trend. `gcc` and `xalanc` exhibit especially high overheads since they use the allocator more frequently than others (as also indicated in Figure 3), which provided the breakdown of various components of *REST*'s slowdown. Especially in the case of `xalanc` which makes a high frequency of allocations (0.2 allocations per kilo-instructions), the allocator overheads dominate significantly compared to other benchmarks. Benchmarks that use the allocator more sparingly (`lbm` and `sjeng`, for instance, which make less than 10 allocation calls overall) have little to negligible overheads.

These results additionally indicate that our allocator, based on ASan, is a major contributor to *REST*'s overhead. This is evidenced by the fact that the full and heap safe categories exhibit almost equal overheads, differing only by 0.16% on average. Thus, if recompilation is an option for users, *REST* could provide stack safety at nominal extra cost. We chose to use the ASan allocator for convenience; in the future, we plan to design a custom *REST* allocator that could potentially mitigate some of the observed overheads.

[5]Weighted arithmetic mean overhead = AriMean(<Plain-normalized runtime>*<Plain runtime>/<Sum of plain runtimes>) - 1

[6]Geometric mean overhead = GeoMean(<Plain-normalized runtime>) - 1

[7]There has been a lot of discussion on the right way of aggregating results [25]. For this work, we follow [26].

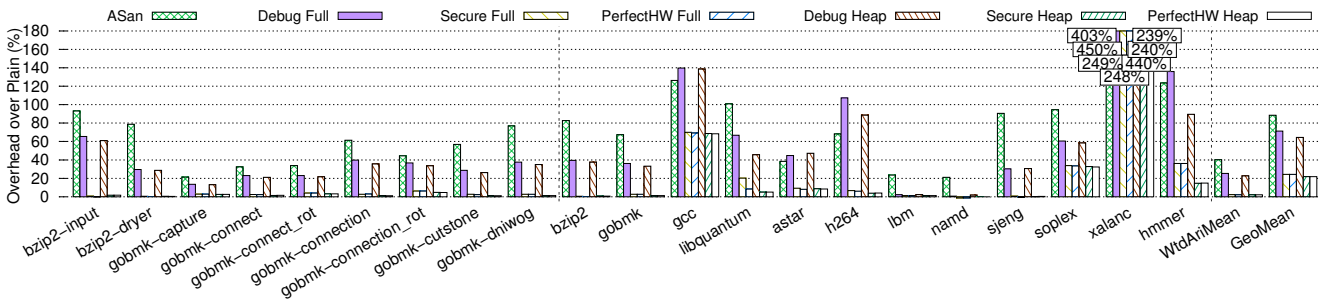

Fig. 7: Runtime overheads of ASan and *REST* in the debug, secure, and perfect (zero-cost) hardware modes while providing full and heap safety. WtdAriMean gives the weighted arithmetic mean overhead, while GeoMean refers to the geometric mean.

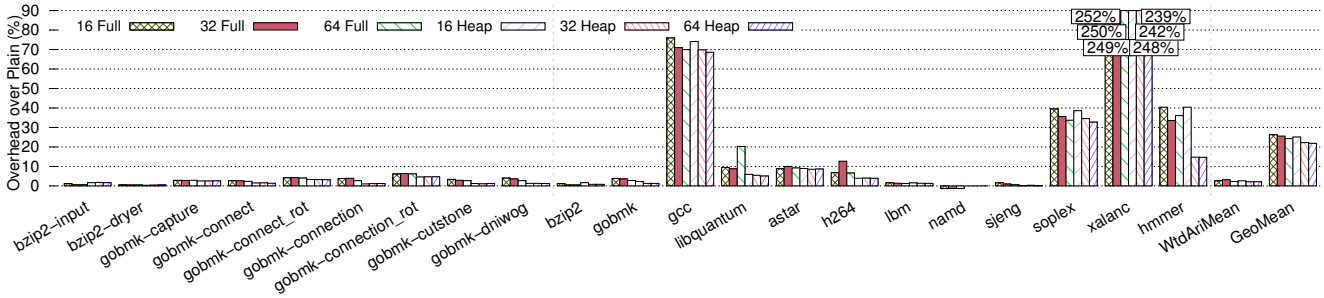

Fig. 8: Runtime overheads of using 16B, 32B and 64B tokens in secure mode. WtdAriMean gives the weighted arithmetic mean overhead, while GeoMean refers to the geometric mean.

The difference in runtimes for the secure and debug modes arises due to the fact that, in the debug mode, we delay store commit until the corresponding write completes. In our simulated out-of-order core, although the impacts of this change manifests in many ways, a few side-effects were predominantly observed. First, unsurprisingly we found that the number of cycles the ROB was blocked by a store was about an order of magnitude higher in the debug mode. IQ occupancy was also severely affected for the latter case, especially for `xalanc` that had the number of cycles IQ was full in the secure and debug modes differed by more than 100x. Notably, we also did not observe a lot of traffic at the main memory interface due to token fills, indicating that most token accesses hit in the cache and do not otherwise contribute to memory access bandwidth for any of the benchmarks in either mode (only 0.04 tokens per kilo-instructions crossed the L2/memory interface for `xalanc` in the secure full run).

Software vs. Hardware. To distinguish between the overheads added by our software and hardware modifications, we run the *REST* binaries on stock hardware with one key modification — each arm and disarm in the binaries is replaced by one regular store. This simulates a situation wherein our *REST* hardware modifications for managing and checking tokens have zero cost. The runtimes for this set of experiments are shown in Figure 7, denoted by the *PerfectHW Full* and *PerfectHW Heap* bars. As these results show, the overheads incurred by the perfect *REST* hardware are not significantly different from that seen in the secure mode, being only 0.2% lower for full protection and only 0.03% lower for heap protection. This implies that the cost of the *REST* primitive in hardware is nearly zero and that the entirety of the performance overheads in the secure mode are solely an artifact of its software component, especially the allocator.

Token Widths. Token widths while affecting the security of a system might also potentially affect its performance, since smaller token widths might allow better cache utilization. In order to evaluate this we configure our implementation to utilize tokens of 16B and 32B and perform the experiment for all modes. The corresponding results are shown in Figure 8. Overall, we see that choosing any single token width does not make a significant difference in terms of performance. In the general case, users might thus freely choose robustness in the form of wider tokens, without compromising performance.

VII. Related Work

Memory safety implies two types of protections — spatial and temporal. Spatial memory errors usually manifest in two different ways depending on program behavior. Overflow-style errors are a result of a sweeping or *linear* access pattern wherein the code sequentially starts accessing locations beyond the bounds of the data structure. Alternatively, invalid reads/writes might also occur if a pointer is corrupted/overwritten resulting in a access pattern that can be more precise or *targeted*. Protection schemes can be characterized depending on which pattern they detect. In terms of temporal protection, schemes can be characterized by the time window within which their protection lasts. Some schemes provide *complete* protection by detecting dangling pointers for the duration of the entire execution, while others only do so until the invalid region has been reallocated again.

Since memory safety has been a persistent problem for decades, a lot of work has been done to address it, especially

Proposal	Spatial Prot.	Temporal Prot.	Shadow Space	Compos-ability	Perf. Overhead	Hardware Modifications
Hardbound [2]	Complete	None	✓	✗	Low	μop injection, L1 cache & TLB for tags
SafeProc [3]	Complete	Complete	✗	✗	Low	Multiple CAMs and memory units, hardware hash table, hash table walker
Watchdog [4]	Complete	Complete	✓	✗	Moderate	μop injection, pointer lock-ID cache, dangling pointer monitor
Watchdoglite [5]	Complete	Complete	✓	✗	Moderate	Nominal
Intel MPX [7]	Complete	None	✗	✗†	High	Not known
HDFI [10]	Linear	None	✓	✓	Negligible	Wider buses and cache lines, tag-aware memory controller with caches, tag table
ADI [27]	Linear‡	Until realloc‡	✗	✓	Negligible	4b per cache line at all cache levels‡
CHERI [6]	Complete	Complete	✗	✗	Moderate	Capability coprocessor tightly integrated with in-order pipeline
iWatcher [28]	N/A	N/A	✗	✓	High	Per-byte cache line metadata, a multi-entry table, small metadata victim cache at L2
Unlimited watch-points [29]	N/A	N/A	✗	✓	High	Range cache, metadata TLB
Safemem [9]	Linear	None	✗	✓	High	Repurpose DRAM's error-correction bits
Memtracker [30]	Linear	Until realloc	✓	✓	Low	Metadata caches, monitoring unit in pipeline
ARM Pointer Authentication [31]	Targeted	None	✗	✓	Negligible	Not known
REST	Linear	Until realloc	✗	✓	Moderate	1 metadata bit per L1-D line, 1 comparator

TABLE III: Comparison of previous hardware techniques (assuming single-core systems for simplicity). †Although MPX-supported binaries execute with modules that are not protected, metadata is dropped when such modules manipulate an MPX-augmented pointer. ‡See text.

in software [32]. In this section, we only discuss relevant hardware techniques proposed towards solving this problem (summarized in Table III) below.

• **Bounds Checking.** Hardware-based bounds checking [2], [3], [4], [5] solutions were proposed to mitigate the problems of high performance overhead associated with software enforced bounds-checking [33], [34], [35] while retaining its effectiveness. They were quite successful in this regard, bringing down the performance penalty significantly (Hardbound [2] reported considerably lower overheads than the others but does not provide temporal safety). There are a few differences between them and *REST*, however. This is because of the fact that while bounds-checking performs complete monitoring of out-of-bounds accesses (assuming pointer identification in hardware is perfect), *REST* only detects errors when the blacklisted locations are accessed and hence, provides weaker security guarantees. The advantages of the latter approach, however, are lower overheads and complexity.

Firstly, *REST*'s memory overhead scales with the number of protected data structures, not pointers to them, and does not need separate memory to do so. We also do not require storage in the chip itself, other than a register at the L1 data cache. On the other hand, most previous works store metadata in a shadow space, a memory region containing metadata for every location of the address space. This results in fast metadata access since calculating its location inside the shadow space is derivable by a simple arithmetic operation on the pointer address. But it is also highly inefficient in terms of storage since all of the address space is shadowed even if a negligible fraction of it is actually occupied by pointers. Watchdog [4] and Watchdoglite [5] reported ~56% increase in memory usage for SPEC CPU benchmarks. In terms of on-chip storage, all schemes, with the exception of Watchdoglite, introduce some form of fast-lookup memory, such as caches, in order to speed up metadata lookup and hence, pointer operations.

Most of these schemes also introduce non-trivial hardware logic to the chip microarchitecture. Hardbound and Watchdog inject micro-op around memory accesses instructions at runtime. SafeProc and Watchdoglite, on the other hand, rely on the compiler to explicitly insert instructions in the program to this end, enabling static analyses to optimize these operations. Furthermore, Watchdog logically extends the physical register file to accommodate metadata, whereas the others use existing registers, thus increasing register pressure. *REST*'s detection logic is vastly simpler since we do not perform checks for spatial and temporal violations in the pipeline for every memory access. Since we defer the detection responsibilities completely to the caches, the core architecture itself remains unchanged, also making register pressure a non-issue.

Additionally, reliance on compiler support implies these systems have limited composability with software (such as third-party libraries) which have not undergone the necessary static transformations. This means they necessarily require shared libraries that have been compiled similarly. Critically however, a kernel that is unaware of this scheme could cause errors and presents a potential vulnerability for such systems.

For instance, an attacker could influence the size arguments of a data-manipulating syscall to corrupt sensitive data. Since *REST* associates metadata with the data structure and not its pointers, we do not have to worry about static pointer analyses (or their accuracy). The compiler support necessary for *REST* is, hence, significantly simpler (LLVM's ASan module has only 2129 LoC with our modifications).

Notably, Intel Memory Protection Extensions (MPX) [7] marks the first commercial support for this technique. However, it faces a few compatibility issues and exhibits high performance overheads [36].

• **Tagging.** Some defenses "color" memory regions by associating tags with them and checking these tags when they are accessed. HDFI [10] marks memory locations with a 1-bit tag, that subsequently indicates whether that location can be accessed via regular load/stores. Although it is quite flexible and exhibits nominal overhead, its hardware requirements are higher than ours. SPARC ADI [27] uses a 4-bit coloring scheme, using the 4 most significant bits of a 64b pointer for this purpose. On an access, the hardware checks whether the tags of the pointer and accessed regions match. They also require a custom allocator responsible for coloring heap allocations but do not require that programs be recompiled to avail this feature. Although full details of the microarchitecture have not been disclosed, at a minimum they require 4 bits of metadata per cache line at all cache levels. Spatial overflows are prevented by annotating adjacent allocations and their metadata with different tags, while temporal overflows are prevented by changing tags on deallocation. However, due to the limited number of available tags, memory regions might reuse tags after being reallocated enough times (via heap feng-shui attacks [37], for instance) after which dangling pointer access will go undetected. Moreover, since they modify pointer format, (legacy) programs that do special pointer operations involving compression or irregular arithmetic will be incompatible with this technology. We do not face these problems.

• **Capabilities.** Capability-based architectures [6], [38] are another metadata-based secure hardware design that offer stronger security guarantees than us. Here, all pointers are augmented with metadata that goes beyond bounds information (permission, for instance). Particularly, works in the CHERI project [6], [39] have demonstrated its applicability in the modern era on a whole-system level, not just for applications, for a MIPS 64-bit in-order processor. However, this support comes at the expense of high performance and area overheads, although the authors acknowledge open areas of optimization in their design.

• **Watchpoints.** This class of solutions aim to provide a high number of hardware data watchpoints, primarily for debugging. iWatcher [28] was one of the first hardware techniques proposed to this end and functionally provided support for a high (but limited) number of programmable hardware watchpoints at a relatively low overhead compared to some software solutions, but required that the affected physical pages be pinned to physical memory and not be swapped out. Although they did not explore memory safety as an application, Greathouse et al. [29] solved both problems by providing unlimited watchpoints and allowed pages to be swapped out by storing metadata separately.

• **Others.** SafeMem [9] repurposed error checking ECC bits in main memory to mark memory locations invalid in order to detect spatial memory errors. They did so by setting the parity state to an error value, so that accesses to those locations trigger exceptions, thus trading reliability for safety. However, each set/unset operation is quite expensive with latencies comparable to an `mprotect` syscall. Additionally, it did not support the swapping main memory contents to disk. Memtracker [30] associates state with each memory location by monitoring accesses to them. They however, do not make any modifications to the allocator to inhibit allocation reuse, and so are more vulnerable to temporal attacks. Besides the above solutions, ARM recently announced pointer authentication in select chips [31] that counter pointer corruption and forging, but do not protect against general temporal or spatial attacks.

VIII. Conclusion and Future Work

In this paper, we proposed *REST*, a primitive for content based checks and showed how it can be used to creae a low complexity, low overhead implementation for improving memory safety. *REST* itself requires local modifications that integrates within existing hardware intefaces. It incurs a low performance penalty for stack and heap safety, which is 22-90% faster than comparable state-of-the-art software implementations, while additionally being more secure and providing heap safety for legacy binaries.

There are many open areas of optimization and extension to *REST*. The *REST* software components viz., the repurposed Address Sanitizer allocator, accounts for almost all of the slowdown in the secure mode. An allocator designed to take advantage of *REST* properties and requirements could be significantly faster. Similarly, for hardware, our goals was to minimize number of optimizations: however, a few additional microarchitectural optimizations such as a dedicated cache for *REST* lines has potential to decrease overheads further, especially for the debug mode and for programs that make frequent allocations. Finally, we only explore *REST* at the application level in this paper; extending and supporting it at the system level and for heterogeneous architectures, will increase system security and reliability.

The benefits of *REST* go well beyond memory safety. As a primitive for performing content-based checks in hardware, it provides a number of opportunities not only for improving other aspects of software security (e.g., control flow), but also programmability and performance. Developing these new applications using *REST* can bring significant exciting benefits.

IX. Acknowledgement

We thank the anonymous reviewers, our colleagues, and other members of the Computer Architecture and Security Technologies Lab (CASTL) at Columbia University, especially Emilio G. Cota, Hiroshi Sasaki, and Adrian Tang, for their valuable feedback and help. This work was supported by HR0011-18-C-0017 (DARPA) and a gift from Bloomberg.

Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government or commercial entities. Simha Sethumadhavan has a significant financial interest in Chip Scan Inc.

REFERENCES

[1] D. Weston and M. Miller, "Windows 10 mitigation improvements," in *Black Hat USA*, 2016.

[2] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the C programming language," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[3] S. Ghose, L. Gilgeous, P. Dudnik, A. Aggarwal, and C. Waxman, "Architectural support for low overhead detection of memory violations," in *2009 Design, Automation Test in Europe Conference Exhibition*, 2009.

[4] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *39th International Symposium on Computer Architecture (ISCA)*, 2012.

[5] S. Nagarakatte, M. Martin, and S. Zdancewic, "WatchdogLite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[6] 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 (ISCA)*, 2014.

[7] Intel Corporation, "Intel architecture instruction set extensions programming reference."

[8] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, "Beyond the PDP-11: Architectural support for a memory-safe C abstract machine," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[9] F. Qin, S. Lu, and Y. Zhou, "SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[10] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[11] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (Usenix ATC)*, 2012.

[12] "Firefox and address sanitizer," https://developer.mozilla.org/en-US/docs/Mozilla/Testing/Firefox_and_Address_Sanitizer.

[13] "Chromium project: Addresssanitizer," https://www.chromium.org/developers/testing/addresssanitizer.

[14] "CVE-2014-0160," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160, 2014.

[15] "AddressSanitizer in hardware," https://github.com/google/sanitizers/wiki/AddressSanitizerInHardware.

[16] "AddressSanitizer Algorithm," https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm.

[17] K. Lu, C. Song, T. Kim, and W. Lee, "Unisan: Proactive kernel memory initialization to eliminate data leakages," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[18] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[19] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[20] G. Novark and E. D. Berger, "DieHarder: Securing the heap," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[21] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[22] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[23] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, 2011.

[25] L. Eeckhout, *Computer Architecture Performance Evaluation Methods*, 1st ed. Morgan & Claypool Publishers, 2010.

[26] L. K. John, "More on finding a single number to indicate overall performance of a benchmark suite," *SIGARCH Comput. Archit. News*, 2004.

[27] "Hardware-assisted checking using Silicon Secured Memory (SSM)," https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html, 2015.

[28] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, "iWatcher: efficient architectural support for software debugging," in *Proceedings of 31st Annual International Symposium on Computer Architecture, 2004.*, 2004.

[29] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin, "A case for unlimited watchpoints," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[30] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "MemTracker: Efficient and programmable support for memory access monitoring and debugging (hpca)," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[31] Qualcomm Technologies Inc., "Pointer authentication on ARMv8.3," https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017.

[32] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*, 2013.

[33] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (Usenix ATC)*, 2002.

[34] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Transactions Programming Language Systems*, 2005.

[35] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[36] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches," *CoRR*, 2017.

[37] A. Sotirov, "Heap feng shui in JavaScript," in *Black Hat Europe*, 2007.

[38] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "CODOMs: Protecting software with code-centric memory domains," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, 2014.

[39] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. Son, M. Roe, S. W. Moore, P. G. Neumann, B. Laurie, and R. N. Watson, "CHERI JNI: Sinking the Java security model into the C," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.