AHEMS: Asynchronous Hardware-Enforced Memory Safety

Kuan-Yu Tseng, Dao Lu, Zbigniew Kalbarczyk, Ravishankar Iyer
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801, USA
mycallmax@gmail.com, {daolu1, kalbarcz, iyer}@illinois.edu

***Abstract*— This paper presents AHEMS (Asynchronous Hardware-Enforced Memory Safety), an architectural support for enforcing spatial and temporal memory safety to protect against memory corruption attacks. We integrated AHEMS with the Leon3 open-source processor and prototype on an FPGA. In an evaluation of the detection coverage using 677 security test cases (including spatial and temporal memory errors), selected from the Juliet Test Suite, AHEMS detected all but one memory safety violation. The missed test case involves overflow of a sub-object in a data structure whose detection is not supported by the current prototype. Performance assessment using the Olden benchmarks shows an average 10.6% overhead, and negligible impact on the processor-critical path (0.06% overhead) and power consumption (0.5% overhead).**

I. INTRODUCTION

Attackers break into computer systems by exploiting security vulnerabilities due to low-level memory corruption errors, e.g., buffer overflow or format strings. Memory errors can be categorized into (i) *spatial errors* which occur when a pointer is used to access memory beyond the bounds of its intended referent (e.g., buffer overflows), and (ii) *temporal errors* which occur when a pointer is used to access an object that no longer exists (e.g., use-after-free or dangling-pointer).

While many defensive techniques against memory corruption attacks have been proposed, most of them are not comprehensive enough. For example, protection schemes that enforce code integrity only prevent code-injection attacks. Defenses that guarantee control-flow integrity can prevent many memory corruption attacks. Unfortunately, they cannot detect non-control-data attacks that do not divert the legal control flow [1].

In this paper, we introduce AHEMS (Asynchronous Hardware-Enforced Memory Safety), which implements *asynchronous memory safety checking*. Asynchronous checking allows AHEMS to offload the bounds-checking operations that occur on each pointer dereference to a security engine, which is a customized hardware implemented as a coprocessor. Since the security engine can be decoupled from the processor chip, AHEMS can reduce the complexity and resource overhead of the main processor. AHEMS places a lightweight runtime monitor in the pipeline of the main processor to facilitate notification of the security engine about memory events (e.g., memory load, store, allocation, and deallocation) and propagate the metadata on each memory pointer. Unlike all other approaches, AHEMS stores the metadata in dedicated physical memory not visible to the main processor. That ensures that the metadata cannot be tampered with by the attacker. *To the best of our knowledge, AHEMS is the first hardware approach that allows completely asynchronous runtime checking to ensure both spatial and temporal memory safety with very low overhead.* Evaluation of an AHEMS prototype on benchmark programs shows an average 10.6% runtime overhead. While the asynchronous property of AHEMS may result in delayed detection of an attack, we show that the detection latency is generally too short to allow an attacker to launch an attack. Furthermore, the latency can be reduced by increasing the size of the FIFO queue that keeps the information on the memory events.

II. RELATED WORK

In this section, we discuss software and hardware approaches that have been proposed to enforce spatial and/or temporal memory safety.

A. Fat-pointer Approaches

SafeC [2], Cyclone [3], CCured [4], and Fail-Safe C [5] use the concept of a *fat pointer* for bound checking. A fat pointer contains not only the address of its intended object but also the base and the bound. It is an efficient data structure that is used by type-safe languages, e.g., Java, to perform bound checking. However, because the representation of a pointer in the memory has changed, fat pointer approaches are not source-compatible [1] and not binary-compatible[2] with precompiled binaries.

B. Object-based Approaches

Object-based approaches maintain an object table that maps an address range to an object. When a pointer is dereferenced, the object table is used to determine the intended target for the bound checking. J&K [6], CRED [7], Mudflap [8], and SAFECode [9], BBC [10], AddressSanitizer [11], and PAriCheck [12] are examples of object-based approaches. The primary limitation of these approaches is very high performance overhead. For

[1] A source-incompatible approach requires manual changes to the source code.

[2] A binary-incompatible means that the protected code cannot interoperate with unprotected code (e.g., precompiled third-party libraries).

example, J&K has 5x to 6x overhead, while CRED has 11x to 12x overhead (on Olden benchmarks). BBC, Address-Sanitizer, and PAriCheck represent another class of object-based approaches that manipulate the arrangement of objects in the memory to improve efficiency. The change of the memory layout makes protected code binary-incompatible with unprotected code, e.g., precompiled libraries. In general, while the object-based approaches detect most spatial memory errors (except sub-object overflows), they provide only limited protection for temporal memory errors. For example, if the intended referent of a pointer `(p)` is freed and subsequent allocation reuses the same memory region as the freed object to create an unrelated object, the access to the unrelated object using `p` causes a temporal memory error that evades the detection of object-based approaches.

C. Pointer-based Approaches

In contrast to object-based approaches, pointer-based approaches (P&F [13], MSCC [14], SoftBound [15], CETS [16], and MemSafe [17]) associate the base and the bound metadata with the pointer instead of the object. Runtime checks are performed only on pointer dereferences, because an out-of-bounds pointer does not harm anything as long as it is not dereferenced. While pointer-based approaches can provide both spatial and temporal memory safety without combining with other methods, their binary compatibility is limited. Since pointer-based approaches need to propagate the base and the bound metadata along with the pointer arithmetic but the propagation of metadata is not performed in the unprotected code, the metadata of a pointer may become outdated after a function call to unprotected code. In contrast, AHEMS provides better binary compatibility in that the metadata can be updated even in the unprotected code, because the propagation is handled by the hardware. Furthermore, AHEMS can perform the bound checks on pointer dereferences of pointers generated by protected code and used in the unprotected code.

D. Hardware Approaches

SafeMem [18] utilizes the ECC bits existing in memory to detect memory leaks and some classes of memory errors. The advantage of SafeMem is that it does not require installation of additional hardware. MemTracker [19] is a hardware-programmable state machine residing in memory that associates each byte of memory with a *state* and treats each access to the memory as an *event*. Both SafeMem and MemTracker fail to detect attacks that allow arbitrary memory writes. Clause et al. [20] associate each pointer and each byte of memory with a *taint mark* and only allow pointers that have the same taint mark to access the memory. Chuang et al. [21] accelerate the metadata lookup of a pointer-based approach similar to SafeC to improve performance overhead. Arora et al. [22] propose an architectural support on embedded processors to reduce the performance overhead of CCured. The two techniques in [20] and [22] employ the fat-pointer approach, which makes them not binary-compatible. SafeProc [23] extends the ISA of a processor with safety instructions and adds architectural support to the processor to detect memory errors. SafeProc also proposes an optimization to delay memory error detection whenever possible, which is similar to the asynchronous property of AHEMS. However, the degree of asynchrony in SafeProc is very limited compared to that of AHEMS, because SafeProc cannot delay the detection if dependency between instructions exists. HardBound [24] is essentially a hardware version of SoftBound that enforces spatial memory safety. Watchdog [25] improves HardBound by implementing a hardware version of CETS that ensures temporal memory safety to achieve temporal and spatial memory safety when combined with HardBound. AHEMS achieves the same level of memory safety and runtime overhead as Watchdog while having additional advantages. For example, AHEMS needs to widen the size of each register by $\log_2 N$ bits (where N is the number of registers in a processor), while Watchdog needs to quintuple the size of each register to keep its metadata. Not only is that a prohibitively high hardware resource overhead to the on-chip memory, but it is lethal to the critical path of the processor. Besides, AHEMS has physical metadata isolation, while Watchdog may suffer from metadata tampering.

III. Approach Overview

Figure 1 illustrates the architecture of AHEMS. The AHEMS framework consists of two major parts: (i) the hardware responsible for the runtime checking of memory safety and (ii) the source-code instrumentation responsible for establishing the interface between the program and the hardware.

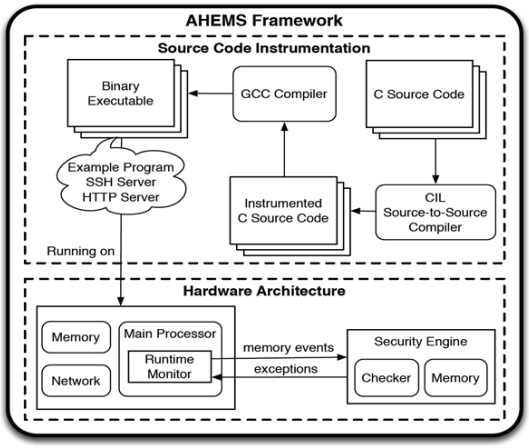

Figure 1: AHEMS Architecture

A. Source Code Instrumentation

A program source code is instrumented with `alloc()`, `dealloc()`, and `subcreate()` functions using the CIL source-to-source compiler [26]. The purpose is to enable the runtime notification of the hardware about memory allocation events, memory deallocation events, and sub-

object creation, respectively (See Figure 2). To track the birth and death of every memory reference, we instrumented (i) the `malloc()` and `free()` functions for tracking dynamically allocated references, (ii) local variables and function parameters, (iii) global and static variables, (iv) the sub-objects in a data structure, and (v) integer-to-pointer casts. The instrumented code is compiled with the GCC compiler to generate a binary executable.

1) `malloc()` and `free()`

AHEMS instruments the `malloc()` and `free()` functions with `alloc()` and `dealloc()`, respectively (see Figure 3). In that way, the hardware is informed about every memory allocation and deallocation event.

2) Local Variables and Function Parameters

Function parameters are treated the same as local variables, and we use local variables to represent both. In order to track local variables, AHEMS (i) inserts an additional local pointer `ptr` for each local array and each local variable whose address is taken, (ii) inserts `alloc()` in the prologue of the function to initialize `ptr` with base and bound, (iii) replaces all uses of the local array with the added local pointer that points to the local array, (iv) replaces each use of a local variable with the added pointer that points to the local variable, and (v) inserts `dealloc()` in the epilogue of the function to mark `ptr` as dead. Note that AHEMS does not track local variables whose addresses are not taken, because their accesses are always memory-safe. Figure 4 shows an example of local variable instrumentation.

3) Global Variables and Static Variables

The tracking of global or static variables is similar to that of local variables except for the insertion positions of `alloc()` functions. There is no need to insert `dealloc()` functions.

4) Sub-objects in the Structure

AHEMS instruments a sub-object of a structure with a `subcreate()` function when the sub-object is an array or the address of the sub-object is taken (thus associating the sub-object with a sub-base and a sub-bound). The lifetime of a sub-object is the same as that of its parent object, so `dealloc()` is not needed for sub-objects. Figure 5 shows an example program before and after the sub-object instrumentation.

5)Integer-to-Pointer Casts

In order to allow integer-to-pointer casts, AHEMS allows programmers to manually insert the `alloc()` function to notify AHEMS that an integer value represents a valid address with a programmer-specified base and bound.

A. Hardware Architecture

There are three main hardware components: *main processor*, *runtime monitor*, and *security engine*. The *main processor* is responsible for program execution. The *runtime monitor* (embedded in the main processor) collects memory events (memory loads, stores, allocations, deallocations, and sub-object creations) and sends these events to the *security engine* for memory safety checking. When a violation of memory safety is detected, the security engine raises an exception, which is communicated to the main processor either to stop the program or invoke an exception handler. The main processor does not need to wait for any information from the security engine except the exceptions, which are rare during normal (i.e., error-free) execution.

```
// return the new pointer to an object associated
// with (base, bound)
void* alloc(void *base, size_t bound);
// mark the reference as a dead object
void dealloc(void *reference);
// Create a pointer to an sub-object associated
// with (subbase, subbound)
void* subcreate(void *subbase, size_t subbound);
```

Figure 2: Definition of Three Instrumentation Functions

```
void *malloc(size_t size) {
  // ... (omitted)
  // compute the return value ret_val
  return alloc(ret_val, size);
}
void free(*ptr) {
  // ... (omitted)
  dealloc(ptr);
}
```

Figure 3: Instrumentation of `malloc()` and `free()`

```
void foo(int in1) {
  int lvar = 5, *ptr, *ptr2;
  int lary[100];


  ptr = &lvar;
  ptr2 = &in1;
  sum = lary[5];
  // use sum and ptr
}
```

(a) Before Instrumentation

```
void foo(int in1) {
  int lvar = 5, *ptr;
  int lary[100];
  int lptr1 = alloc(&lvar, 4);
  int lptr2 = alloc(lary, 40);
  int lptr3 = alloc(&in1, 4);
  ptr = lptr1;
  ptr2 = lptr3;
  sum = lptr2[5];
  // use sum and ptr
}
```

(b) After Instrumentation

Figure 4: An Example of Local Variable Instrumentation

```
struct Node {
  int nm[10];
  int *ptr;
  int value;
};
void foo() {
  struct Node n;


  n.nm[5] = 3;
}
```

(a) Before Instrumentation

```
struct Node {
  int nm[10];
  int *ptr;
  int value;
};
void foo() {
  struct Node n;
  struct Node *np;
  int (*nmp)[10];
  np = alloc(&n, sizeof(n));
  nmp = subcreate(&np->nm,40);
  (*nmp)[5] = 3;
}
```

(b) After Instrumentation

Figure 5: An Example of Sub-object Instrumentation

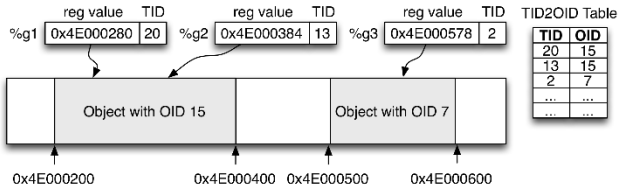

Figure 6: Relation among Register Values, TIDs, and OIDs

1) New machine instructions

To implement the `alloc()`, `dealloc()`, and `subcreate()` functions needed to track pointers, AHEMS adds three new machine instructions, `alloc`, `dealloc`, and `subcreate`, to the main processor.

2) Temporary Identifier and Object Identifier

Each register in the main processor is associated with a temporary identifier (TID). If a register represents a pointer in the program, its TID is used to identify the object to which the register is pointing. Specifically, the security engine uses the TID to look up the object identifier (OID), which is unique for an object, in the security engine's lookup table, `TID2OID`. Figure 6 illustrates the relationship among the register values, the TIDs, and the OIDs. If two registers point to different objects, they have different OIDs in the `TID2OID` table. Conversely, if two registers point to the same object, they may have different TIDs, but they have the same OID even if their offsets into the object are different. For example in Figure 6, registers %g1 and %g2 point to different offsets of the same object with OID = 15; register %g3 points to a different object with OID = 7. Note that although %g1 and %g2 have different TIDs, they are linked to the same OID in the TID2OID table. When a register is used as an address for a `load` or `store` instruction, the security engine can find the object's `(base, bound)` to check whether the pointer is within legal range by going through the lookup process `tid→oid→(base, bound)`.

3) TID and OID Generation and Propagation Rules

For each of the six operations in the main processor (memory allocations/deallocations/loads/stores, pointer arithmetic, and sub-object creations), additional actions need to be executed along with the original operation to maintain the relationship between OID and TID and to check memory safety. Table I summarizes those actions and their actors. We describe each of the actions in detail below.

Memory allocation (see "`alloc` Instruction" in Table I). When an object (or a region of memory) is allocated, `alloc` is executed (i) to associate the pointer pointing to the object with a new TID (Step 1 in Table I); (ii) to notify the security engine to assign a new OID to that object and map the new TID to the new OID in the `TID2OID` table (Step 2 in Table I); and (iii) to associate the `(base, bound, NOPARENT)` information of the object with the OID in the `OID2BaseBound` table (where `NOPARENT` means that this object is not a sub-object). These actions provide the *mother pointer*[3] that points to an object with a new TID. The security engine can use this TID to look up the status of an object using the `TID2OID` and `OID2BaseBound` tables when the pointer is dereferenced.

[3] The pointer obtained at the time memory was allocated for an object.

Pointer arithmetic (see "Pointer Arithmetic" in Table I)**.** To ensure that each pointer is associated with the correct object, AHEMS needs to propagate the TID when the pointer arithmetic is performed. For example, if the instruction "`ADD rd, rs, imm`" is performed (i.e., `rd = rs + imm`) and `rs` represents a pointer to an object, the destination register `rd` should also point to the same object as `rs`. Hence, the TID of `rs` should be copied to that of `rd`. Since source code instrumentation enforces the requirement that all subsequent pointers to an object be derived from the *mother pointer* to the object, every subsequent access to the object can always use the propagated TID to look up the OID in the `TID2OID` table to check the object's liveness and boundary.

TABLE I: TID AND OID GENERATION AND PROPAGATION RULES

`alloc` Instruction	**Actor**
`ALLOC rd, base, bound`	Main Processor
`1. rd.tid = tid = tid_gen()`	Runtime Monitor
`2. TID2OID[tid] = oid = oid_gen()` `3. OID2BaseBound[oid] = (base, bound, NOPARENT)`	Security Engine
Pointer Arithmetic	**Actor**
`ADD rd, rs1, rs2`	Main Processor
`if rs1.tid != INVALID then rd.tid = rs1.tid` `else rd.tid = rs2.tid`	Runtime Monitor
`ADD rd, rs1, imm`	Main Processor
`rd.tid = rs1.tid`	Runtime Monitor
`MOV rd, rs`	Main Processor
`rd.tid = rs1.tid`	Runtime Monitor
`dealloc` Instruction	**Actor**
`DEALLOC rd`	Main Processor

`1. oid = TID2OID[tid]`
`2. OID2BaseBound[oid] = INVALID`
`3. TID2OID[tid] = INVALID`
	Security Engine

`subcreate` Instruction	**Actor**
`SUBCREATE rd, subbase, subbound`	Main Processor
`1. rd.tid = tid = tid_gen()`	Runtime Monitor

`2. parent_oid = TID2OID[subbase.tid]`
`3. TID2OID[tid] = oid = oid_gen()`
`4. OID2BaseBound[oid]=(subbase,subbound,parent_oid)`
	Security Engine

`store` Instruction	**Actor**
`STORE [addr], rs`	Main Processor

`1. addr_oid = TID2OID[addr.tid]`
`2. (base,bound,parent_oid)= OID2BaseBound[addr_oid]`
`3. if (base,bound,parent_oid) = INVALID`
`then raise exceptions`
`4. if (parent_oid != NOPARENT and`
`OID2BaseBound[parent_oid]==INVALID)`
`then raise exceptions`
`5. if (addr < base) or addr > (base + bound)`
`then raise exceptions`
`6. MEM2OID[addr] = TID2OID[rs.tid]`
	Security Engine

`load` Instruction	**Actor**
`LOAD rd, [addr]`	Main Processor
`1. rd.tid = tid_gen()`	Runtime Monitor

`2. addr_oid = TID2OID[addr.tid]`
`3. (base,bound,parent_oid)= OID2BaseBound[addr_oid]`
`4. if (parent_oid != NOPARENT and`
`OID2BaseBound[parent_oid] == INVALID)`
`then raise exceptions`
`5. if (base, bound) = INVALID`
`then raise exceptions`
`6. if (addr < base) or addr > (base + bound)`
`then raise exceptions`
`7. TID2OID[rd.tid] = MEM2OID[addr]`
	Security Engine

Memory deallocation (see "`dealloc` Instruction" in Table I). When an object is freed, `dealloc` is executed to invalidate the object so that any subsequent access to the object triggers an alarm. Specifically, `dealloc` notifies the security engine to mark `TID2OID[tid]` and `OID2BaseBound[oid]` entries as invalid.

Sub-object creation (see "`subcreate` Instruction" in Table I). Creation of a pointer to a sub-object is similar to memory allocation except that the sub-object should also be associated with the OID of its parent object (steps 2 and 4 in Table I) so that the security engine can use the OID of its parent object (denoted by `parent_oid`) to determine the lifetime of the sub-object on loads and stores by checking whether `OID2BaseBound[parent_oid]` is `INVALID`. Therefore, the OID of the sub-object is associated with `(base, bound, parent_oid)` instead of `(base, bound, NOPARENT)` in the `OID2BaseBound` table.

Store (see "`store` Instruction" in Table I). Two additional actions are required along with a `store`: (i) checking the liveness and the boundary of the accessed object, and (ii) mapping in the `MEM2OID` table the memory address (`addr`) to the OID of the object to which the register `rs` points. For the first action, the information on the accessed object is found through the following lookup process: `addr.tid` → `oid` → `(base, bound)` (steps 1 and 2 in Table I). Then the validity of `(base, bound, parent_oid)` is checked to see whether the object is still live (steps 3 and 4 in Table I). Finally, `addr` is compared with the `(base, bound)` to see whether the memory address is within the range of the object (step 5 in Table I). If the object is not live or the `addr` is out of bounds, the security engine raises an exception. For the latter action (step 6 in Table I), the `MEM2OID` table allows the security engine to recover the mapping between TID and OID when the register value being stored in the memory is loaded back to the register. (See the next paragraph for details.)

Load (see "`load` Instruction" in Table I). Actions needed along with `load` involve checking the liveness and boundary of the accessed object in the same way as `store` does. In addition, two other actions are performed: (i) association of the destination register (`rd`) with a new TID (step 1 in Table I) and (ii) association of the new TID with the OID of the pointer stored at the memory address `addr` (step 7 in Table I). The `load` instruction loads the pointer stored at `addr` into the destination register (`rd`). Those two actions recover the association between the stored pointer and the object to which the stored pointer points. That way AHEMS keeps track of a pointer even if the pointer is stored in the memory and loaded back for later use.

IV. IMPLEMENTATION

Figure 7 illustrates the implementation of AHEMS on top of the open-source Leon3 architecture [27]. Leon3 is a 32-bit processor compliant with the SPARC V8 instruction set. The *runtime monitor* is embedded in the *main processor* to generate and propagate the TID. It also sends the memory events to the *security engine* via a FIFO buffer with configurable size. The *security engine* maintains the metadata, checks memory safety violation, and raises an exception to notify the *runtime monitor* about any violation.

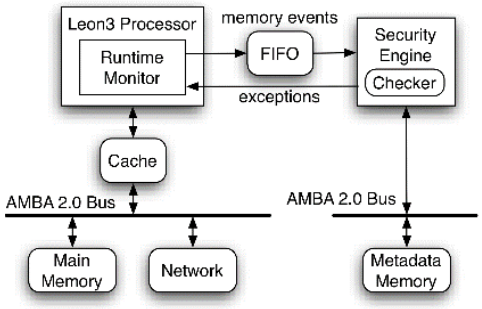

Figure 7: AHEMS on Leon 3 Architecture

Main Processor. When pointer arithmetic such as `rd = rs + 8` is executed, the TID of `rs` needs to be copied to the TID of `rd` (`rd` and `rs` are destination and source registers, respectively). To support TID propagation, each register in the main processor is widened by $\log_2 N$ bits to record the associated TID (N is the total number of registers). Accordingly, the ALU of the Leon3 processor and the instructions used for pointer arithmetic are also changed. We modified the Leon3 processor to treat `cpop2` with `opc=0x01`, and `opc=0x02`, `opc=0x03` as `alloc`, `dealloc`, and `subcreate` instructions, respectively. Normally the Leon3 processor treats `cpop2` as a `nop` instruction. Note that our prototype does not support `subcreate`.

Runtime Monitor. The *runtime monitor* is responsible for TID generation and data transmission to the security engine. A new TID is generated and associated with the destination register when a `load` or `alloc` instruction is executed. The Leon3 processor is connected to the security engine through a FIFO with configurable size. The size of the FIFO affects the detection latency of the memory safety violations. Each entry of the FIFO contains the metadata of a memory event. Figure 8 shows the FIFO entry format for memory events, where `newtid` is the TID generated by the TID Generator; `rs.tid` is the TID associated with the source register of the store instruction; `addr` is the address value of the `load` or `store` instruction; `addr.tid` is the TID associated with the address; `PC` is the address of the instruction.

Security Engine. The security engine is implemented as a state machine responsible for checking memory safety and maintaining the consistency of the `TID2OID`, `OID2BaseBound`, and `MEM2OID` tables. All three tables reside in the dedicated memory of the security engine. The `TID2OID` contains 136 entries (the number of registers in the main processor). Each entry is 32 bits wide and stores an

OID. The OID is directly used as the address to locate the *base* and *bound*. Each entry of the OID2BaseBound table is 8 bytes (4 bytes each for the base and bound). The MSB (Most Significant Bit) of the bound is used to indicate whether an entry is valid. Each entry of the MEM2OID table stores an OID (32 bits wide).

The actions executed along with each instruction (i.e., alloc, dealloc, load, or store) to complete the memory safety checks are implemented by hardware state machines in the security engine. Some actions require memory access, while others only need register file access. Table II summarizes the execution time and memory overhead of actions performed for each instruction.

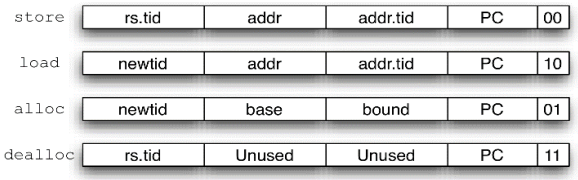

Figure 8: Format of FIFO Entry for Memory Events.

TABLE II: TIME OF ACTIONS PERFORMED FOR EACH INSTRUCTION

Instruction Type	Execution Time
load instruction	5 cycles + 2 memory accesses
store instruction	4 cycles + 2 memory accesses
alloc instruction	3 cycles + 1 memory accesses
dealloc instruction	4 cycles + 2 memory accesses

V. EXPERIMENTAL EVALUATION

We synthesized the AHEMS prototype and ran it on an FPGA board Xilinx Virtex-5 FXT ML510 with 80 MHz CPU frequency and two DIMMs of 512 MB DDR2 memory. Note that our prototype does not instrument global or static variables and does not detect sub-object overflows.

Detection Coverage. To evaluate the detection coverage of the AHEMS prototype, we employed the Juliet Test Suite version 1.2 provided by NIST [28]. The Test Suite contains a total of 61,387 test cases from 118 different CWEs (Common Weakness Enumerations) for C/C++. We assessed the AHEMS prototype by running representative test CWEs cases that can potentially lead to memory corruption attacks. In particular, the selected CWEs assess defenses against stack/heap-based overflow, integer overflow, use-after-free, double-free, and dangling pointers. We have tested a total of 677 test cases and only one case went undetected (see Table III). The undetected test case overflows a sub-object in a structure to overwrite the pointer in the *next* field of the structure. Although sub-object overflow detection is currently not supported by our AHEMS prototype that is not a limitation of the AHEMS architecture.

Runtime Overhead on the Olden Benchmark. We measured the runtime performance overhead of our AHEMS prototype on Olden benchmarks [29]. The benchmarks contain pointer-intensive programs with an average of 711 LOCs (Lines of Code), and many researchers have used Olden benchmarks for performance evaluation (e.g., [9] [24]). Figure 9 shows the runtime overhead of AHEMS on Olden benchmarks. AHEMS exhibits, on average, 10.6% runtime overhead (with a maximum of 38.4%). For most of the programs in Olden, the security engine overlaps memory safety checking with main processor execution time very well, so the runtime overhead is low. However, for programs that are memory-intensive, such as *bisort*, the security engine may not have enough time to perform memory safety checking. The main processor stalls waiting for the security engine, because the consumption rate of memory events (from the FIFO buffer) by the security engine is lower than the production rate of memory events by the main processor.

TABLE III: CWES FROM JULIET TEST SUITE TESTED ON AHEMS

Spatial Memory Errors			
CWE No.	**Description**	**Tested**	**Detected**
CWE121	Stack-based Buffer Overflow	209	208
CWE122	Heap-based Buffer Overflow	18	18
CWE124	Buffer Underwrite	102	102
CWE126	Buffer Overread	145	145
CWE127	Buffer Underread	33	33
CWE588	Attempt to Access Child of Non-structure Pointer	34	34
CWE680	Int. overflow to Buffer Overflow	38	38
CWE761	Free Ptr Not at Start of Buffer	38	38
Subtotal		**617**	**616**
Temporal Memory Errors			
CWE No.	**Description**	**Tested**	**Detected**
CWE415	Double-free	38	38
CWE416	Use-after-free	20	20
CWE562	Return of Stack Variable Address	2	2
Subtotal		**60**	**60**
Total		**677**	**676**

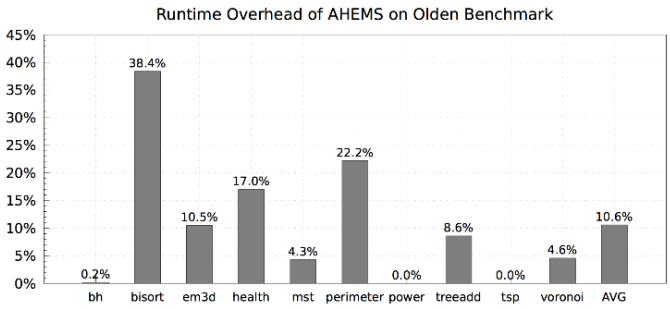

Figure 9: AHEMS Runtime Overhead

Performance Comparison with Other Approaches. In order to compare AHEMS with state-of-the-art memory safety approaches, we ran the following four approaches on Olden benchmarks: Mudflap [8], Softbound+CETS [15] [16], SAFECode [9], and AddressSanitizer [11]. We chose to run the measurements on an x86 Linux machine because (i) that is the primary platform for software-based approaches, and (ii) most of these approaches are not stable and not fully optimized for the SPARC platform. Consequently, testing

those methods on the SPARC platform could potentially bias the results to our advantage. In Table IV, the runtime overhead of the AHEMS prototype is compared with the other four approaches. It can be seen that the runtime overhead of AHEMS is more than an order of magnitude lower than that of any of the other approaches. AHEMS has an average of 10.6% overhead, while Mudflap, Softbound+CETS, SAFECode, and AddressSanitizer have 17,420%, 381%, 340%, and 144% overhead, respectively.

TABLE IV: COMPARISON OF RUNTIME OVERHEAD

Programs	AHEMS	Mudflap	Softbound+CETS	SAFECode	AddressSanitizer
Bh	0.2%	13655.2%	Compiler error	Runtime error	41.9%
bisort	38.4%	3114%	341.1%	154.3%	74.7%
em3d	10.5%	705.0%	473.0%	192.1%	89.5%
health	17%	13343.9%	737.8%	943.2%	361.5%
Mst	4.3%	1169.2%	395.1%	644.1%	92.2%
perimeter	22.2%	32317.8%	443.1%	212.7%	142.8%
power	0.0%	263.9%	1.2%	1.0%	2.7%
treeadd	8.6%	106008%	448.1%	518.8%	398.7%
tsp	0.0%	1404.9%	206.9%	57.5%	76.8%
voronoi	4.6%	2224.2%	False alarm	Runtime error	156.5%
Average	**10.6%**	**17420%**	**381%**	**340%**	**144%**

For hardware approaches, Arora [22] reports 10% and 100% overhead on `em3d` and `health`, respectively. That is higher than the AHEMS overhead corresponding to those two programs. SafeProc [23] achieves 9% overhead on average, which is slightly lower than AHEMS's overhead. However, it requires nontrivial source code modification, which may impede its deployment. Hardbound [24] has an average of 9% overhead; however it only enforces spatial memory safety. Watchdog [25] reports an average runtime overhead of 24% on 20 SPEC benchmarks. SafeMem [18] has 6.3% overhead on average on nonstandard benchmarks, and MemTracker [19] has 2.7% on SPEC benchmarks. However, their memory safety protection is less comprehensive than that of AHEMS. For example, SafeMem does not detect attacks that allow arbitrary memory writes, while MemTracker fails to detect the buffer overflow that overwrites decision-making variables on the stack to launch non-control data attacks.

Critical Path. We synthesized the Leon3 system with and without AHEMS to compare the post-place-and-route static timings using the Xilinx ISE tool. The delays on the critical path with and without AHEMS were 12.47 ns (equivalent to 80.192 MHz) and 12.463 ns (equivalent 80.238 MHz), respectively. AHEMS has a negligible (0.06%) impact on the critical path of the Leon3 system.

Hardware Resource Overhead and Power Consumption. We estimated the hardware resource and power consumption overhead of the AHEMS prototype by comparing the results from the Leon3 system synthesized with and without AHEMS. AHEMS adds about 13.8% overhead in terms of occupied FPGA slices. In terms of power consumption, AHEMS has 94.3% and 22.3% overhead on signals and logic, respectively. Those estimates were obtained using the Xilinx XPower Analyzer. However, the total power consumption overhead is only 0.5%, because signals and logic contribute to a very small portion of the total power consumption.

Detection Latency. The downside of an asynchronous approach is that detection of memory violations may be delayed. Increasing the size of the FIFO buffer used to keep the data on memory events improves the performance of AHEMS. We estimated the minimum (empty FIFO) and maximum (full FIFO) detection latency of AHEMS. Since load and store instructions occupy most of the FIFO entries (`alloc` and `dealloc` are much less frequent), we used the average time needed to process `load` and `store` instructions to approximate the detection latency. Specifically, we calculated the detection latency of AHEMS on the Intel Core 2 Duo E6400 processor with a 2.13 GHz clock. The specifications of the system, including IPC (instructions per cycle) and cache miss rate, are listed in Table V. Using the values in Table V and Table II, we get:

$$MemLatency = 5.7\ (cycles)$$
$$DetectLatency_{empty} = 16\ (instructions)$$
$$DetectLatency_{full} = 16817\ (instructions)$$

To launch an attack, an attacker has an average time window corresponding to the execution of 16 instructions and 16,817 instructions when the FIFO is empty and when the FIFO is full, respectively. It takes 7.66 μs to execute 16,817 instructions, which is too short to launch an attack in practice, even if it is fully automated. For example, a call to a *libc* function `system(" ")` with an empty string as a command takes more than 1 ms on a machine with the Intel Core i5-3470 3.2 GHz CPU. That means that a shell cannot be launched within 7.66 μs. More importantly, the FIFO size is configurable, and system designers can adjust the size of the FIFO based on the tradeoff between performance and detection latency.

TABLE V: SPECIFICATIONS OF AN INTEL CORE 2 PROCESSOR-BASED SYSTEM RUNNING SPEC2006 BENCHMARK

Attribute	Value
Memory	2GB (1GB*2) DDR2 533MHz
L1/L2 Cache Hit Time	3 cycles/12 cycles
L2 Cache Miss Penalty	165 cycles
IPC[4]	0.97
L1/L2 Miss Rate	2.25%/0.41%
AHEMS FIFO Size	1024 entries * 16 bytes/entry

VI. CONCLUSIONS

This paper presents AHEMS, an architectural support for asynchronous checking to ensure both spatial and temporal memory safety. Evaluation of the prototype AHEMS implementation shows an average 10.6% overhead on Olden benchmarks, and negligible impact on the processor's critical path (0.06% overhead) and power consumption (0.5% overhead).

[4] We use SPEC2006 benchmarks as the representative programs to approximate the IPC and cache miss rates of real-world programs.

VII. ACKNOWLEDGMENTS

This work was supported in part by the Department of Energy under Award Number DE-OE0000097, by the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement No. FA8750-11-2-0084, and the Defense Threat Reduction Agency under award number HDTRA1-11-1-0008. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the sponsoring organizations.

REFERENCES

[1] S. Chen, X. Jun, E. Sezer, P. Gauriar and R. Iyer, "Non-control-data attacks are realistic threats," in *USENIX SecuritySymposium*, 2005.

[2] T. M. Austin, S. E. Breach and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1994.

[3] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney and Y. Wang, "Cyclone: a safe dialect of C," in *Proc. of the General Track of the USENIX Annual Technical Conf.*, 2002.

[4] G. C. Necula, J. Condit, M. Harren, S. McPeak and W. Weimer, "CCured: type-safe retrofitting of legacy software," *ACM Trans. on Prog. Lang. and Sys.*, vol. 27, no. 3, 2005.

[5] Y. Oiwa, "Implementation of the memory-safe full ANSI-C compiler," in *Proc. of the 2009 ACM Conf. on Prog. Lang. Design and Impl.*, 2009.

[6] R. W. M. Jones, P. H. J. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *Third Int'l Workshop on Automated Debugging*, 1997.

[7] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *Proc. of the 11th Net. and Dist. Sys. Sec. Symp.*, 2004.

[8] F. C. Eigler, "Mudflap: pointer use checking for C/C++," in *GCC Developer Summit*, 2003.

[9] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *Proc. of the Int'l Conf. on Software Engr.*, 2006.

[10] P. Akritidis, M. Costa, M. Castro and S. Hand, "Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors," in *Proc. of the 18th Conf. on USENIX Sec. Symp.*, 2009.

[11] K. Serebryany, D. Bruening, A. Potapenko and D. Vyukov, "AddressSanitizer: a fast address sanity checker," in *Proc. of the USENIX Annual Technical Conf.*, 2012.

[12] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens and W. Joosen, "PAriCheck: an efficient pointer arithmetic checker for C programs," in *Proc. of the 5th ACM Symp. on Info., Comput. and Comm. Sec.*, 2010.

[13] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in C," *Softw. Pract. Exper.*, 27(1), 1997.

[14] W. Xu, D. C. DuVarney and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of C programs," in *Proc. of the 12th ACM SIGSOFT Int'l Symp. on Foundations of Software Engr.*, 2004.

[15] S. Nagarakatte, J. Zhao, M. Martin and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for c," in *Proc. of the ACM Conf. on Prog. Lang. Design and Impl.*, 2009.

[16] S. Nagarakatte, J. Zhao, M. Martin and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *Proc. of the Int'l Symp. on Memory Management*, 2010.

[17] M. S. Simpson and R. K. Barua, "MemSafe: ensuring the spatial and temporal memory safety of C at runtime," *Softw. Pract. Exper.*, 43(1), 2013.

[18] F. Qin, S. Lu and Y. Zhou, "SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs," in *11th Int'l Symp. on High-Perf. Comput. Archit.*, 2005.

[19] G. Venkataramani, I. Doudalis, Y. Solihin and M. Prvulovic, "MemTracker: An accelerator for memory debugging and monitoring," *ACM Trans. Archit. Code Optim.*, 6(2), 2009.

[20] J. Clause, I. Doudalis, A. Orso and M. Prvulovic, "Effective memory protection using dynamic tainting," in *Proc. of the 22nd Int'l Conf. on Automated Software Engr.*, 2007.

[21] W. Chuang, S. Narayanasamy and B. Calder, "Accelerating Meta Data Checks for Software Correctness and Security," *Journal of Instruction-Level Parallism*, vol. 9, 2007.

[22] D. Arora, A. Raghunathan, S. Ravi and N. K. Jha, "Architectural support for safe software execution on embedded processors," in *Proc. of the 4th Int'l Conf. on Hardware/Software Codesign and System Synthesis*, 2006.

[23] S. Ghose, L. Gilgeous, P. Dudnik, A. Aggarwal and C. Waxman, "Architectural support for low overhead detection of memory violations," in *Proc. of the Conf. on Design, Auto. and Test in Europe*, 2009.

[24] J. Devietti, C. Blundell, M. M. K. Martin and S. Zdancewic, "Hardbound: architectural support for spatial safety of the C programming language," in *Proc. of the 13th Int'l Conf on Archit. Support for Prog. Lang. and Operating Sys.*, 2008.

[25] S. Nagarakatte, M. Martin and S. Zdancewic, "Watchdog: hardware for safe and secure manual memory management and full memory safety," in *Proc. of Int'l Symp. on Comput. Archit.*, 2012.

[26] G. C. Necula, S. McPeak, S. P. Rahul and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proc. of the 11th Int'l Conf. on Compiler Construction*, 2002.

[27] Aeroflex Gaisler AB, *Leon3 Processor.*

[28] NIST, "Juliet Test Suite for C/C++".

[29] M. C. Carlisle, "Olden: parallelizing programs with dynamic data structures on distributed-memory machines," PhD Thesis, Princeton University Department of Computer Science, June 1996.