ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/317915136

Shakti-T: A RISC-V Processor with Light Weight Security Extensions

Conference Paper · June 2017

DOI: 10.1145/3092627.3092629

CITATIONS
37

READS
4,050

5 authors, including:

Arjun Menon
InCore Semiconductors Pvt. Ltd.
6 PUBLICATIONS **76** CITATIONS
SEE PROFILE

M. Subadra
Indian Institute of Technology Madras
10 PUBLICATIONS **68** CITATIONS
SEE PROFILE

Chester Rebeiro
Indian Institute of Technology Madras
79 PUBLICATIONS **717** CITATIONS
SEE PROFILE

Neel Gala
Indian Institute of Technology Madras
20 PUBLICATIONS **142** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

SHAKTI Microprocessor View project

SHAKTI Processor Project View project

All content following this page was uploaded by Arjun Menon on 18 November 2017.

The user has requested enhancement of the downloaded file.

Shakti-T : A RISC-V Processor with Light Weight Security Extensions

Arjun Menon , Subadra Murugan , Chester Rebeiro
Neel Gala , and Kamakoti Veezhinathan
Department of Computer Science and Engineering
Indian Institute of Technology Madras, India
{*c.arjunmenon, mail2subadram, chetrebeiro, neelgala, veezhi* }*@gmail.com*

ABSTRACT

With increased usage of compute cores for sensitive applications, including e-commerce, there is a need to provide additional hardware support for securing information from memory based attacks. This work presents a unified hardware framework for handling spatial and temporal memory attacks. The paper integrates the proposed hardware framework with a RISC-V based micro-architecture with an enhanced application binary interface that enables software layers to use these features to protect sensitive data. We demonstrate the effectiveness of the proposed scheme through practical case studies in addition to taking the design through a VLSI CAD design flow. The proposed processor reduces the metadata storage overhead up to 4× in comparison with the existing solutions, while incurring an area overhead of just 1914 LUTs and 2197 flip flops on an FPGA, without affecting the critical path delay of the processor.

CCS Concepts

•Security and privacy → Hardware security implementation; •Computer systems organization → Reduced instruction set computing;

Keywords

Secure Processor Architecture, Buffer Overflow, Memory Security, Tagged Architecture, Spatial Attacks, Temporal Attacks

1. INTRODUCTION

The biggest security threats to computing systems today comprise attacks that exploit vulnerabilities within the memory subsystem. These attacks can either be spatial or temporal in nature. Spatial attacks occur when a particular pointer accesses memory regions beyond its permissible range. Temporal attacks, on the other hand, occur when a pointer accesses a memory region that has been freed after allocation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HASP '17, June 25, 2017, Toronto, ON, Canada

© 2017 ACM. ISBN 978-1-4503-5266-6/17/06…$15.00

DOI: http://dx.doi.org/10.1145/3092627.3092629

One of the most popular form of spatial memory attacks is *buffer-overflow* [28]. The first buffer-overflow vulnerability was witnessed in the *Morris worm* in 1987 which enabled corruption of data by allowing malicious code on the stack to execute. Researchers have also found several ways to exploit this vulnerability, such as the *blaster worm* [5] and the *slammer worm* [21] which have been used to perform Distributed Denial of Service attacks within a network.

The first line of defense against the buffer-overflow vulnerability was to ensure that the stack content is no longer executable [35]. As the manifestations of buffer-overflow evolved over time, such as *return-to-libc* [30] and *Return Oriented Programming* (ROP) [29], several software defined solutions came into existence. Some of the proposed solutions include: stack canaries [8]; encryption of the code pointer [9]; storing the return address in a shadow stack [11, 33, 12]; re-arranging argument locations, return addresses, previous frame pointers and local variables [34]; control flow integrity checks [1]; and, Address Space Layout Randomization (ASLR) [31]. Though literature shows that ASLR has proven to be the most effective and widely adopted technique for protection against ROP based attacks, there do exist well known attacks that either leak sensitive information [32] or change the control flow [18] by bypassing ASLR.

A promising solution capable of fending a vast variety of buffer-overflow manifestations is the *fat-pointer* solution. *Fat-pointer* derives its name from the fact that every pointer is associated with a base and a bound which together define the permissible region of memory access. Thus, each time a pointer is dereferenced, the effective address is checked to ensure it lies within the base and bound values. This solution has also been extended to prevent *read-buffer-overflow* based attacks such as *Heartbleed* [15] and *format-string-attacks* [26]. There exists numerous software solutions [3, 19, 25, 39, 2] which implement fat-pointers for security. However, these solutions either incur high runtime overhead or lack the ability to provide a robust solution against all known buffer-overflow manifestations. Hardware solutions[14, 37, 27], on the contrary, incurs minimal run-time overheads in addition to providing strong security guarantees.

Fat pointers require the base and bounds associated with every pointer to be stored in memory thereby increasing storage overhead. A lightweight hardware implementation of fat-pointers (Low-Fat pointers), as proposed in [20], reduces this storage overhead by encoding the base and bounds into a custom 16-bit floating point format which is stored in the upper bits of the 64-bit virtual address. The floating point encoding, however, loses granularity for higher nu-

merical values of the base and bounds thereby leading to memory fragmentation (since only certain memory sizes can be defined using limited granularity offered by the floating point format).

Temporal attacks, on the contrary, are not as prevalent as spatial attacks, but recently many instances of temporal attacks have come into existence. Some prominent forms of temporal attacks include *use-after-free* [38, 7] and *double-free*[18] attacks which exploit dangling pointers to corrupt memory regions which have been freed by an aliased pointer. One of the recent works, Watchdog [23] and WatchdogLite [24], provide security against temporal in addition to spatial memory attacks. This is achieved by associating an identifier with every pointer and using a *lock-and-key* mechanism similar to the ones proposed in software [22, 4]. Watchdog and WatchdogLite also perform bounds checking by employing fat-pointers which lead to total storage overhead of 320-bits for every non-aliased pointer declared. This overhead multiplies significantly when strong pointer aliasing is present within an application.

A common design choice amongst hardware implementations employing fat-pointers is to use a shadow register-file to store the base and bounds values thereby reducing memory accesses. Considering that at any point only a small fraction of the architectural register-file contain pointers rather than data, the shadow register-file remains highly under-utilized. Furthermore, while aliased pointers point to the same memory region, each pointer holds its own copy of the base and bound values which not only increases the redundant storage overheads but also leads to temporal vulnerabilities in the memory region. For *e.g.*, consider a set P of aliased pointers $\{P_1, P_2, ..., P_n\}$ pointing to the same memory region. If at any instant a pointer P_i has freed the memory or has performed a memory re-allocation then the base and bound values of only P_i are updated while the base and bounds of the remaining $(n-1)$ pointers in P remain the same, thereby pointing to an invalid memory region. This particular scenario can lead to several temporal and spatial vulnerabilities to exist within the memory.

In view of the above stated challenges and drawbacks associated with modern day secure hardware implementations, we propose Shakti-T[1], a RISC-V [36] based 64-bit processor encompassing a set of performance efficient extensions which provide strong security guarantees against both temporal and spatial memory attacks while incurring minimal runtime and memory storage overheads. The novel contributions of Shakti-T processor include:

- Security against all types of spatial memory attacks while incurring minimal overheads in terms of storage and performance.
- Defense against temporal memory attacks by eliminating dangling pointers in events of memory release or re-allocation.
- Reduced metadata storage overheads (of the order of $2\times$ and $4\times$ over [14] and [23, 24] respectively) by using a common memory region across all pointers to store the base and bounds.
- An in-core Content Addressable Memory which stores the base and bounds to reduce the runtime overhead of fetching them from memory. This data structure implementation is far more efficient than the conventional shadow registers employed in literature.

The rest of the paper is organised as follows. Section-2 discusses the hardware extensions proposed in Shakti-T. Section-3 elaborates on the Instruction Set Architecture (ISA) extensions required for Shakti-T. Case studies describing how the hardware and the ISA extensions of Shakti-T work collectively to prevent attacks under different scenarios are presented in Section-4. Section-5 highlights the micro-architectural implementation details of Shakti-T. The results and overhead analysis of Shakti-T are presented in Section-6. Section-7 concludes the paper.

2. HARDWARE EXTENSIONS

While Shakti-T uses the conventional concept of fat-pointers to enable security against spatial memory attacks, it differs from traditional architectures in its implementation of the concept; which in turn provides aid to thwart temporal memory attacks. In Shakti-T the base and bounds for all the pointers are stored in a separate memory region - *Pointer Limits Memory* (*PLM*) which resides in the main memory. The base address of the PLM is stored in a special register called the *Pointer Limits Base Register* (*PLBR*). To link the pointer to its respective base and bound values, every pointer is associated with a pointer-id (`ptr_id`) which acts as the offset into the PLM region, and is stored alongside the pointer in the memory. Thus, every time a pointer is dereferenced, the location of its base and bound values is obtained by adding its `ptr_id` to the PLBR register.

This implementation approach provides a forthright advantage of storage efficiency in case of aliased pointers. For instance, consider a set of n aliased pointers $\{P_1, P_2, ..., P_n\}$, all pointing to the same memory region. Under this scenario, the solution proposed in [14, 37] would require a storage overhead of $2n$ words (base and bounds for each pointer), while the solution proposed in [23, 24] would incur a storage overhead of $4n + 1$ words (base, bound, lock and key per pointer; and a single key value for the entire set). Shakti-T, on the other hand, requires to store only $n + 2$ extra words (one `ptr_id` per pointer and a single base and bound value for the entire set) demonstrating significant reduction in storage overheads as compared to [14, 37, 23, 24] which are shown in Section-6. If at any instant a pointer P_i releases the memory or re-allocates the memory region, the base and bounds in the PLM are updated with new values. This update is seen by all the pointers in their next access thus preventing invalid access or temporal attacks by the other aliased pointers.

While Shakti-T clearly offers benefits in scenarios where aliased pointers exist, in scenarios where all pointers point to different memory regions it incurs relatively higher storage overheads (three words per pointer - `ptr_id`, base and bound) as compared to [14, 37] which incurs an overhead of only two words per pointer (base and bound). This overhead can be justified by the additional implicit security guarantee provided by the PLM against temporal memory attacks, which is absent in [14, 37].

An obvious run-time overhead of the above proposed scheme is the increase in number of memory accesses required to fetch the pointer-id, the base value and the bound value each time a pointer is loaded into a register from the memory. To address this overhead we pro-

[1]The proposed solution involves the use of *Tag-bits* and has been applied to the open-source C-64 processor of *Shakti* [17]; hence, the name Shakti-T.

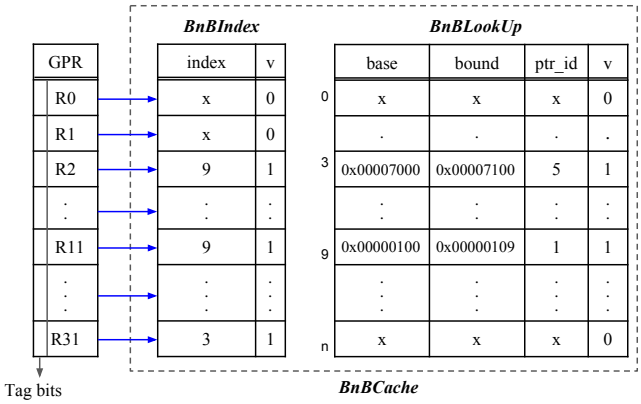


Figure 1: Architecture of the Base and Bound Cache (BnBCache)

pose an on-chip hardware design extension called the *Base-and-Bound Cache*(BnBCache) as shown in Figure-1 which aids in the reduction of the total number of memory accesses. The BnBCache contains two internal data structures: the BnBIndex table and the BnBLookUp table. The BnBLookUp table holds the base and bound values required by the pointers, while the BnBIndex table holds the index of the BnBLookUp table which contains the relevant base and bound values for the concerned pointer. The detailed working of each of the modules of Figure-1 is presented below. We make an assumption here that, like any other tagged architecture used for security against spatial attacks, each 64-bit in the memory is associated with a single *Tag-Bit* indicating whether the content is a pointer (if Tag-bit = 1) or regular data (if Tag-bit = 0).

1. Architectural Register-File. Each architectural register, i.e. General Purpose Register (GPR), holds a Tag-bit which indicates that the content in the respective register is a pointer or a regular data. The Tag-bits are set by the compiler and stored alongside the respective memory.

2. BnBLookUp. The BnBLookUp is a special memory array, where each entry of the array holds 4 fields: the *base* value (64-bits), the *bound* value (64-bits), the `ptr_id` (64-bits) and a *valid* bit. The depth of BnBLookUp is a design choice and can be decided based on the maximum number of active pointers at any given instant by executing a set of benchmark programs in the intended application domain.

A value of 1 in the valid field indicates that the memory region defined by the respective base and bound values is a live memory region. On the other hand, a value of 0 in the valid field can either indicate an unused entry in the BnBLookUp (after system-reset) or can indicate that the memory region corresponding to base and bound values have been released by a pointer.

3. BnBIndex. The BnBIndex table, in the conventional sense, can be considered to be a shadow register-file *i.e.,* each architectural register is associated with a single entry of the BnBIndex. The BnBIndex holds two fields: an index and a valid bit. If the valid bit of a particular entry is 1, it implies that base and bound values of the particular pointer are present in the BnBLookUp table. On the contrary, a value of 0 in the valid field indicates that the base and bounds for the pointer needs to be fetched from the memory and stored in the BnBLookUp table. Once the BnBLookUp table has been populated, the index of the entry being updated in the BnBLookUp is stored in the index field of the BnBIndex table and the valid bit is set to 1.

If at any instant, an entry, say i, of the BnBLookUp table is deleted or overwritten with new data, all the entries in the BnBIndex table containing the value i in the index field are invalidated by resetting their valid bit to 0 to avoid misreading the base and bound values. This mechanism further ensures that if a particular pointer releases its memory, then all the aliased pointers present in the architectural register-file will be notified about this action implicitly thereby preventing temporal attacks due to freed pointers.

Each time a pointer is loaded from a memory location (m_i) into one of the architectural registers (GPR[i]), the `ptr_id` of that pointer (pt_i) is fetched from address $m_i + 8$. Subsequently, pt_i is compared to the `ptr_id`s of all the valid entries in BnBLookUp table to find a match. If a match occurs, then the index of that BnBLookUp entry is stored in the respective entry of the BnBIndex. However, if none of the entries match, the base and bound values are fetched from memory locations pt_i and $pt_i + 8$ respectively, and stored in an unused entry of the BnBLookUp table. Also, the valid bit of the respective BnBLookUp table is set to 1. If there are no unused entries in the BnBLookUp to store the newly fetched base and bounds, then an entry (say j) is replaced using a pseudo random policy (other replacement policies such as LRU, Pseudo-LRU, etc can be explored as well). This replacement further causes all the entries in the BnBIndex table pointing to entry j to be invalidated by resetting their valid bits to 0.

Side Effects. The above described hardware extensions lead to certain side-effects which need to be handled separately. For instance, consider a function *fn1* being called within an application. Each time this function is called, the registers used by the function are pushed on to the stack and before exiting the function, the values are popped back into the respective registers. During this operation it is possible that one of the registers, say GPR[i], being pushed onto the stack, holds a pointer (P_i). This pointer also holds valid corresponding entries in the BnBIndex and BnBLookUp table. Let us consider the case that during the execution of the function *fn1*, these entries were overwritten with some new values due to new pointer initializations within the function. During function epilogue, P_i is popped back from the stack and written into GPR[i]. However, the base and bounds values for this pointer are no longer available in the BnBLookUp table and neither does a mechanism exist to retrieve them from the PLM.

In order to address this issue, we propose to maintain an auxiliary stack - BnBStack (Base and Bound Stack) - which stores the pointer-ids of the pointers being pushed-in and popped-out during a function call. This BnBStack is accessed using a special register called BnBSP (BnB Stack Pointer). Thus, when a register that holds a pointer (Tag-bit=1) is being written onto the stack, then the corresponding `ptr_id` from the BnBCache is also stored onto the BnBStack using BnBSP. Similarly, when a value is being read from the stack to a register, if the Tag-bit of the data is set, then the `ptr_id` from the top of the BnBStack is also read and written into the BnBCache (if an entry of the `ptr_id` does not already exist). This `ptr_id` is then used to fetch the values of base and bounds if they are not already present in the BnBLookUp table. Additional instructions required

to maintain these data structures in a coherent state are presented in Section-3.

Design Rationale. Compared to most of the traditional implementations of fat pointers, where the base and bound values are stored in an auxiliary register alongside every architectural register, we propose a disjoint structure (BnBCache) which gives certain advantages over the former design choice. Firstly, correlating the base and bounds along with the architectural register introduces unnecessary dependencies between instructions which only update either the base and bound values, or the value of the register. Having a decoupled structure would eliminate stalls that occur due to these false dependencies. Secondly, since only a fraction of the registers hold pointers at a given instant in time, most of the auxiliary registers would remain unused; whereas having a unified structure would result in a better utilization of these registers.

2.1 Illegal pointer-arithmetic.

A tagged architecture, in general, is more robust to ensure type safety and other properties. Works such as [37] and [13] have exploited Tag-bits to enforce fine-grained information flow control and also enable better debugging and instrumentation of programs. Authors of [10, 16] have demonstrated how tag bits can be used to enforce information flow control and semantics of programming languages. In Shakti-T, we exploit this additional leverage offered by Tag-bits to enforce certain rules on pointer-arithmetic. To demonstrate this additional boon, we have implemented the following set of rules in the hardware that the pointer-arithmetic need to comply with to avoid generating an exception. Here, T_1 and T_2 refer to the tag bits of operand1 and operand2 respectively.

1. For a load or store operation, T_1=1.
2. If T_1=1 and T_2=1, only subtraction and store operations are allowed.
3. If T_1=1 and T_2=1, and subtraction operation is performed, the tag of the result is set as 0.
4. If T_1=0, and T_2=1, then only addition operation is valid.
5. If T_1=1, and T_2=0, the instruction should either be an addition, subtraction, load or store instruction.
6. If T_1=1, and T_2=0, all instructions except loads and stores are valid.

Rules 1, 4 and 5 are specific to the ISAs where the loads and stores always happen with respect to a base register, which in turn are always passed through operand1. While, we have implemented only the above six rules, several other rules can be explored and added to Shakti-T.

3. ISA EXTENSIONS

The previous section described the various hardware additions such as the Tag-bit, PLBR, BnBIndex, BnBCache and the BnBSP required in order to safeguard against spatial and temporal attacks. However, each of these hardware structures need to be initialized and managed in a coherent fashion to avoid exceptions in hardware. In order to facilitate the management of these resources, we propose a set of instructions which enable proper functioning of different hardware blocks, thereby providing strong security guarantees. These instructions will be added by the compiler, and no source code changes are required, thereby enabling an easy adoption of the scheme.

This section describes the proposed set of instructions in detail. Here, *rs1*, *rs2* and *rs3* refer to source GPRs, *rd* refers to the destination GPR, and, *imm* refers to the immediate value. Except for *wrtag*, *wrspreg* and *rdspreg* instructions, whose immediate value field is 1-bit, remaining instructions have a 12-bit value which is consistent with the RISC-V ISA.

1. Write Register Tag [`wrtag rd, imm`]:
Definition: This instruction writes the value of *imm* (0 or 1) in the tag-field of a GPR indexed by *rd*.
Usage: When a pointer is assigned a value (using either *malloc* or *realloc* functions), this instruction is used to set the Tag-bit of the register holding the pointer to 1. Similarly, when a dynamically allocated memory is being freed (using the *free* function), the Tag-bit of the register pointing to this memory region is set to 0 using this instruction.
Side-effect: Whenever the Tag-bit of a register is reset (i.e., a value 0 is written) the corresponding entry in the BnBIndex is invalidated. Simultaneously, all entries in the BnBIndex holding the same index value as that of register *rd* are also invalidated. Additionally, the entry in the BnBCache holding the base and bounds is also invalidated (by setting the valid bit to 0) and a store request is generated to clear the entries in the PLM corresponding to the pointer-id of *rd*. Thus, if any aliased pointer pointing to the same base and bound values is accessed, the processor raises an exception indicating a memory access violation.

2. Write Special Registers [`wrspreg rs1, imm`]:
Definition: This instruction loads a value into one of the special registers (PLBR or BnBSP). If *imm* is 1, then the contents of GPR[rs1] are written into the BnBSP register; otherwise, then the contents of GPR[rs1] are written into the PLBR register.
Usage: This is a privileged instruction (i.e., can execute in only supervisor or hypervisor modes) issued by the Operating System in-order to assign a PLM region and a BnB_Stack to a program before it is loaded for execution. Using conventional immediate arithmetic or load instructions, the value of GPR[rs1] is initialized, following which *wrspreg* is issued. Also, during a context switch, this instruction is used to load the new PLBR and BnBSP values.

3. Read Special Register [`rdspreg rd, imm`]:
Definition: If *imm* is 1, the contents of BnBSP are written into GPR[rd]; otherwise the contents of PLBR are written into GPR[rd].
Usage: Similar to *wrspreg*, this too is a privileged instruction. When a context switch occurs, this instruction is used to store the value of PLBR and BnBSP into memory along with rest of the process state. This instruction copies the special register contents into a GPR which is subsequently stored in memory using a conventional store instruction.

4. Write PLM [`wrplm rs1, rs2, rs3`]:
Definition: This instruction populates the PLM memory region with the base and bound values. Register *rs1* holds the `ptr_id` of a pointer, while registers *rs2* and *rs3* hold the base and the bound values respectively. Thus, the contents of *rs2* and *rs3* are stored at memory locations pointed by PLBR + GPR[rs1] and PLBR + GPR[rs1] + 8 respectively.

Usage: This instruction is issued when a new memory region is allocated either on the stack (new array declaration), or on the heap (new pointer declaration).

5. Load Base and Bounds [`ldbnb rd, rs1`]:

Definition: This instruction loads the base and bound values from the PLM into the BnBCache. Since GPR[rs1] holds the `ptr_id`, the base and bounds can be fetched from the memory address PLBR + GPR[rs1].

Usage: In a typical program, local arrays are accessed using the stack pointer whose base and bound values will be the complete stack frame. Therefore, to ensure that the accesses of local variables in the stack are legal, we introduce the `ldbnb` instruction, which loads the base and bound values of a local variable to the BnBCache. One would argue that this instruction can be treated as a side-effect of the *wrplm* instruction, i.e., while writing into the PLM, we can simultaneously update BnBCache and BnBIndex. This, however, will lead to populating the BnBCache even with the base and bounds of those pointers which are only loaded to GPRs to copy its value to another pointer. In order to prevent pollution of the BnBCache, we propose two separate instructions *ldbnb* and *wrplm* to populate the BnBCache and the PLM respectively.

6. Load Pointer [`ldptr rd, rs1, imm`]:

Definition: This instruction reads the pointer located at the address *rs1+imm* and loads it into the register *rd*. Subsequently, the `ptr_id` of the pointer is fetched from address *rs1+imm+8* to check if the base and bound values associated with this pointer are present within the BnBCache. If not, they are fetched from the memory (effective address= `ptr_id` + PLBR) and stored in the BnBCache.

Usage: This instruction is used when an already initialized pointer is loaded back from the memory to the GPR.

7. Function store [`fnst rs1, imm(rs2)`]:

Definition: This instruction stores the contents of register *rs1* to the memory address *rs2+imm*. If *rs1* happens to be a pointer, then the corresponding `ptr_id` from BnBCache is stored in the location pointed by BnBSP-8 and the BnBSP is decremented by 8.

Usage: This instruction is used to replace the conventional push/store operations performed during a function call to save the contents of registers onto the stack. Typically, *rs2* will hold the stack pointer. This instruction prevents the scenario described in the last section where a pointer popped from the regular stack may lose its pointer-id, and, base and bound values after a function executes.

8. Function load [`fnld rd, imm(rs1)`]:

Definition: This instruction loads the contents of the register *rd* with data present in the memory location pointed by *rs1 + imm*. If the memory location holds a pointer (indicated by the Tag-bit), then the pointer-id, located at the location pointed by BnBSP is read, and used to populate the BnBCache with the respective base and bound values if they are absent. Subsequently, the BnBSP is incremented by 8.

Usage: Complementary to the previous instruction (fnst), this instruction is used to load the pointer into a GPR and its corresponding pointer-id, and base and bound values into the BnBCache just before a function exit.

4. CASE STUDIES

In this section we provide a sample C-code and explain how the ISA extensions mentioned in the previous section can be utilized to safeguard the program from various vulnerabilities. A sample C code snippet is shown in Figure-2(a). It should be noted here that all the cases of Figure-2(a) (i.e. Cases 1 to 5) are all part of the same program, and have been segregated into multiple segments for ease of explanation. Figure 2(b) shows the stack organisation of the complete program.

The program is written using RISC-V [36] assembly, and hence the RISC-V defined architectural registers are used. In the RISC-V ISA, *sp* is the stack pointer, *t0*, *t1* and *t2* are temporary registers, and, *a0* and *a1* are argument registers. According to the RISC-V calling conventions, these argument registers are used to send and receive arguments to and from a function. Also, for ease of readability the newly added instructions have been italicised and colored in blue in Figure 2(a).

Code Assumptions:
The character arrays *a* and *b* have been allocated 10 and 20 bytes respectively, and all the pointer variables have been assigned 16B (128-bits) of memory (which are byte-addressable) on the stack. The first 8B of the pointer hold the value of the pointer, and the next 8B hold the `ptr_id` of the pointer. For example, the value of pointer *p* is stored from address *sp* + 80 to *sp* + 87, and its `ptr_id` is stored from *sp* + 88 to *sp* + 95. Also assume that the variables *a*, *b*, *ptr1*, and *ptr2* are assigned `ptr_id`s 1, 2, 3, and 4 respectively. Pointers *p* and *q*, being aliased pointers, are not assigned a new `ptr_id` at compile time because they are never dynamically allocated and will therefore get the same pointer-id of an existing pointer.

The following paragraphs describe the various cases of Figure-2(a) in detail:

Case 1. The contents of array *a* need to be copied into array *b* using the *strcpy* function. The conventional compiler generates instructions to set the GPRs *a0* and *a1* with the base addresses of arrays *a* and *b* respectively. This is performed by instruction 1.1 for array *a*, and instruction 1.6 for array *b*.

In Shakti-T, every array has a `ptr_id` associated with it, which in turn points to the base and bounds of the array. When a character array is declared, the values of base and bounds need to be initialized as well, and this is done by instructions 1.2 to 1.5. Since *a* is passed as an argument to the *strcpy* function as a pointer, instruction 1.5 loads the base and bounds (BnB) corresponding to the pointer. Likewise instructions 1.7 to 1.10 do the same for variable *b* and then the *strcpy* function is invoked. Note that since values of `ptr_id`s are known during compile time, instructions 1.2 and 1.7 initialize the `ptr_id`s using an immediate instruction provided in the RISC-V ISA.

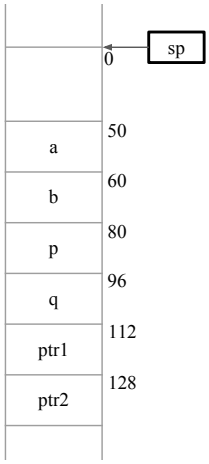
Case 2. In this case, a character pointer *p* points to the array *a*. The variables *p*, its `ptr_id`, *q* and its `ptr_id` are stored in the stack at an offset address of 80, 88, 96, and 104 respectively. The conventional compiler computes the base address of *a* as sp + 50 (instruction 2.1), and stores that value at the address sp + 80 (instruction 2.2). In Shakti-T, the `ptr_id`s also need to be copied which is fulfilled using instructions 2.3 and 2.4. Likewise, instructions 2.5 to 2.8 initialise pointer *q* to point to array *b*.

	Case 1	Case 2	Case 3	Case 4	Case 5

C code	char a[10], b[20];
	…
	strcpy (b,a);
		char *p, *q;
		p= &a;
		q=&b;
			if(...)
			p=q;
			a[7]= *(p+3);
				char *ptr1= (char *)
				malloc (10* sizeof
				(char))
					char **ptr2= &ptr1;
					…
					free(ptr1);

Assembly code	1.1 addi a0, sp, 50; 1.2 *addi t0, zero,1;* 1.3 *addi t1, a0, 9;* 1.4 *wrplm t0, a0, t1;* 1.5 *ldbnb a0, t0;* 1.6 addi a1, sp, 60; 1.7 *addi t0, zero, 2;* 1.8 *addi t1, a1, 19;* 1.9 *wrplm t0, a1, t1;* 1.10 *ldbnb a1, t0;* // … call strcpy;	2.1 addi t1, sp, 50; 2.2 st 80(sp), t1; 2.3 *addi t0, zero, 1;* 2.4 *st 88(sp), t0;* 2.5 addi t1, sp, 60; 2.6 *addi t0, zero, 2;* 2.7 st 96(sp), t1; 2.8 *st 104(sp), t0;*	if(...) 3.1 ld t0, 96(sp); 3.2 st 80(sp), t0; 3.3 *ld t0, 88(sp);* 3.4 *st 104(sp), t0;* 3.5 *ldptr t0, 80(sp);* 3.6 ld t1, 3(t0); 3.7 st 57(sp), t1;	// call malloc; 4.1 *addi t0, a0, 9;* 4.2 *addi t1, zero, 3;* 4.3 *wrplm t1, a0, t0;* 4.4 *wrtag a0,1;* 4.5 st 112(sp), a0; 4.6 *st 120(sp), 3;*	5.1 addi t0, sp, 112; 5.2 *addi t1, t0, 15;* 5.3 *addi t2, zero, 4;* 5.4 *wrplm t2, t0, t1;* 5.5 *wrtag t0,1;* 5.6 st 128(sp), t0; 5.7 *st 136(sp), t2;* … // call free; 5.8 *addi t0, zero, 3;* 5.9 *wrtag t0,0;*

(a)



(b)

Figure 2: (a) Example C code snippets and their corresponding generated assembly code. The italicised and the blue-font refer to the newly inserted instruction. (b) Stack organisation of the program.

Case 3. This case emulates a conditional execution statement which cannot be resolved at compile time. The conventional compiler generates code that conditionally copies the value of q to p (instructions 3.1 and 3.2). In Shakti-T, instructions (3.3 and 3.4) are inserted to copy the `ptr_id` of q to p.

When the pointer p is used to access data, a normal `ld` instruction which loads the pointer into a GPR (in this example *t0*) is replaced with the `ldptr` instruction (instruction 3.5). This instruction fetches the pointer and its corresponding `ptr_id`, and populates the BnBCache with the required base and bound values if not already present. When pointer p is dereferenced, instruction 3.5 fetches the data from memory and instruction 3.6 assigns the fetched value to $a[7]$. Note that when instruction 3.6 executes, the effective address computed is $3 + t0$, and this address is checked by the hardware to be within the pointer's base and bounds.

Case 4. This case explores the scenario where pointers are initialized dynamically using functions such as *malloc, realloc*, etc. Here, the pointer *ptr1* is stored at an offset of 112 in the stack, and its `ptr_id` at an offset of 120. The *malloc* function returns the base address of the pointer through the argument register *a0*. Instruction 4.1 sets the value of the bound in register *t0*, and instruction 4.2 sets the value of `ptr_id` of *ptr1*. The base and bound values are also stored in the PLM by instruction 4.3. This is followed by instruction 4.4 which sets the Tag-bit of the pointer as 1. Finally, the pointer is stored back into the stack by instruction 4.5 along with its `ptr_id` (instruction 4.6).

Case 5. This example describes the necessary code insertion required when a pointer is used to point to an existing pointer. Here to begin with, the *ptr1*, its `ptr_id`, *ptr2* and its `ptr_id` are stored in the stack at an offset of 112, 120, 128 and 136 respectively. When *ptr2* is assigned the address of *ptr1*, the base and bound values of *ptr2* are set as 112 and 127 respectively, and this is done by instructions 5.2-5.4. Additionally, a `wrtag` instruction (instruction 5.5) is issued to set the tag bit of ptr2 to 1. Finally, the value of *ptr2* is stored in the memory, along with the value of its `ptr_id` using instructions 5.6 and 5.7 respectively. When the memory location is freed, a `wrtag` instruction (instruction 5.9) is issued to set the tag bit of the pointer to 0, thereby clearing the value of base and bounds in the PLM and the BnBCache as well.

5. MICRO-ARCHITECTURE

The hardware and ISA extensions proposed in the previous sections have been implemented over an existing baseline processor in order to provide a fair comparison of the incurred area and performance overheads. We have used the 64-bit 5-stage in-order Shakti C-64 design [17] as our baseline processor whose micro-architecture is shown in Figure-3 (non-colored blocks refer to the design of the baseline processor). Following is a brief outline on the functioning of Shakti C-64:

1. Fetch Stage: This stage generates a new Program Counter (PC) and fetches the relevant instruction from the Instruction cache (I-cache). The fetched instruction is then stored in the IF-ID Inter-Stage Buffer (ISB).

2. Decode Stage: This stage reads the instruction from the IF-ID ISB, decodes it to identify the type of instruction, the operand register addresses, the destination register address, etc. and stores this information in the ID-EXE ISB.

3. Execute Stage: This stage fetches the operands from the register-file and executes the instruction using the ALU (Arithmetic-Logic Unit). If the instruction is a branch instruction which generates a jump (i.e. branch is taken) then the IF-ID ISB is invalidated and the PC is set to the new address of the branch. In case of a memory instruction, this stage simply calculates the address of the memory access. The result of the execution is stored in the EXE-MEM ISB.

4. Memory Stage: If the instruction executed does not require access to the memory/cache then it is simply buffered into MEM-WB ISB. However, in case of a load/ store instruction the request is sent to the data cache to perform the necessary operation. On completion of the memory/-cache access, the information is stored in the MEM-WB ISB.

5. Write-Back Stage: This is the final stage of the processor which commits the result into the register-file if no exception was generated for that instruction in any of the

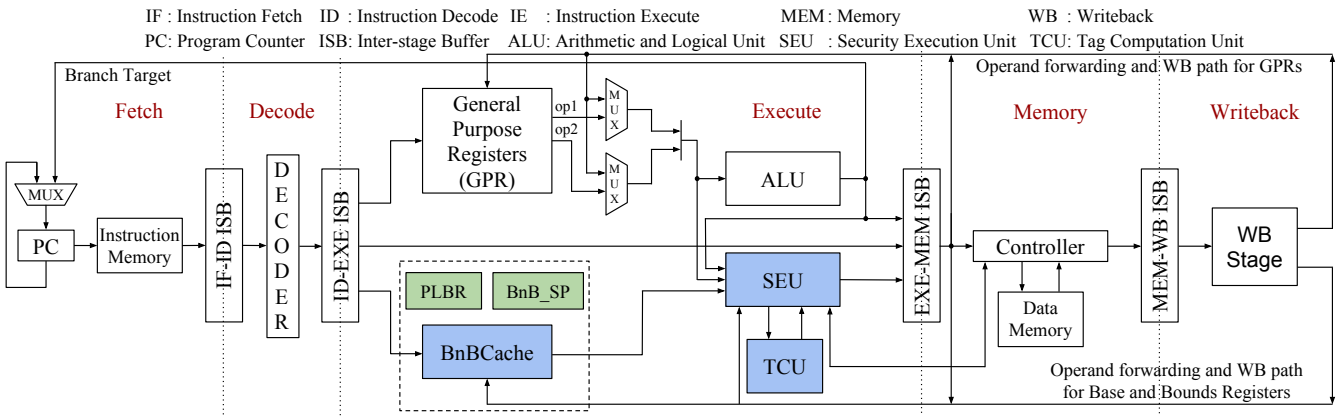


Figure 3: Micro-architecture of Shakti-T. Non-colored blocks refer to components of the baseline processor.

previous stages. In case a trap is generated, all the ISBs are invalidated and the PC in the fetch stage is set to the address of the specific Interrupt Service Routine (ISR).

The colored components of Figure-3 indicate the necessary hardware blocks that have been added to C-64 to implement Shakti-T. Following is a brief outline of the added blocks:

1. BnBCache: This block is integrated into the Execute stage of the C-64 baseline processor. The functioning of this component has been explained in detail in Section-2. The BnBCache is accessed whenever the base and bound values need to be read or modified using the instructions mentioned in Section-3.

2. Security Execution Unit (SEU): This unit is also a part of the execution stage which executes all the new instructions proposed in Section-3. The result of the computation is stored in the EXE-MEM ISB.

3. Tagged Computation Unit (TCU): This unit is implemented as an extension of the SEU unit in the Execute stage. This unit enforces the pointer-arithmetic semantics described in Section-2.1 and computes the Tag-bit of the result.

In addition to the above mentioned major changes, the decoder and the memory controller have also been slightly modified to decode the newly proposed instructions and communicate with the above mentioned blocks.

6. RESULTS

The Shakti-T micro-architecture of Figure-3 has been implemented using Bluespec-System-Verilog (BSV) [6]. The instruction and data caches are implemented as Block-RAMS (BRAMs). Each 64-bit word of the data-cache has an extra Tag-bit associated with it. This leads to an overhead of 1.6% in size of the data-cache. While this seems as an obvious overhead over the baseline processor, in a generic tagged architecture employing security features such as fine-grained control flow integrity, compartmentalization, etc., the cost of the Tag-bit will be amortized.

The BnBIndex and the BnBLookUp tables are both implemented as arrays of registers. For comparison purposes we have designed the BnBLookUp table with 8 entries. Since the RISC-V ISA mandates 32-GPRs, the BnBIndex has 32 entries each holding a valid bit and a 3-bit value to index into the BnBLookUp Table. Each entry of the BnBLookUp table holds a 64-bit pointer-id, 64-bit base value, 64-bit bound value and a valid bit. Accounting for the extra Tag-bit associated with every GPR, the BnBCache structure of Figure-1 requires a total of 1704 flip-flops along with the associated control circuitry as an overhead.

Shakti-T, with the above design choices, and C-64 have been synthesized in ASIC using UMCIP's open 55nm library through Synopsys Design Compiler^{TM}, and also for a Virtex Ultrascale FPGA (*xcvu095-ffva2104-2-e*) using Xilinx Vivado 2016.1. The BRAMs of the caches for the ASIC flow are treated as black-boxes for both C-64 and Shakti-T. Shakti-T incurs an area overhead of 1914 LUTs and 2197 flip flops on the FPGA, and 11834 cells in ASIC synthesis. For both the synthesis flows, i.e. FPGA and ASIC, the proposed hardware does not fall in the timing critical path of Shakti-T. The 2197 flip-flops not only include the BnBIndex and BnBLookUp table entries but also the extra flip-flops required in the ISBs to store/forward the base and bound values. It should be noted, though we have chose C-64 as the baseline processor, the mentioned overheads are absolute in nature and will incur the same overhead for any other baseline processor. Moreover, since the proposed solution does not require any changes to the source code, this system can be widely deployed with minimal effort.

Table-1 provides a summary of the overall comparison of Shakti-T with previous works in terms of the type of security guarantee, the instrumentation involved, the overheads, etc.

7. CONCLUSION

In this paper we propose Shakti-T - a lightweight security extension for defense against both temporal and spatial memory attacks. Shakti-T employs the concept of base and bounds to ensure that pointers access valid memory regions. As compared to conventional methods, which store the base and bounds for every pointer explicitly in memory, Shakti-T associates a pointer-id with every pointer which in turn points to a common memory region (PLM) which holds the base and bound values for all pointers. This mechanism avoids redundant copies of the same base and bound values and also reduces the storage overhead per pointer. To further reduce the overheads of fetching these base and bounds from the memory, Shakti-T adopts an in-core hardware block (BnBCache) which stores the base and bound values of few active pointers. Shakti-T also proposes a set of instruction extensions required to maintain a coherent state of the BnBCache data structure. Unlike some of the existing works, the solution of Shakti-T does not require any modifi-

Table 1: Comparison with previous works

		[27]	[14]	[20]	[23]	[24]	Shakti-T
Safety Checking	Spatial	✓	✓	✓	✓	✓	✓
	Temporal	✗	✗	✗	✓	✓	✓
Bounds Check Instrumentation	Compiler	✓	✗	✗	✓	✓	✗
	Hardware	✗	✓	✓	✓	✗	✓
Metadata size		128 $\times n$	128 $\times n$	0	256 $\times n + 64$	256 $\times n$ +64	64 $\times n$ +128
Fragmentation		✗	✗	✓	✗	✗	✗
Perf. Overhead	HW	NA	NA	5%	NA	NA	0%
	SW	NA	10%	NA	25%	29%	NA

cations of the application source code being targeted for the Shakti-T processor. The proposed solution (implemented on top of a RISC-V ISA based 64-bit baseline processor) incurs an area overhead of just 1914 LUTs and 2197 flip flops on an FPGA without any increase in the critical path delay. As an additional security benefit, the architecture of Shakti-T can be used to enforce semantic rules to avoid illegal pointer arithmetic.

8. REFERENCES

[1] M. Abadi et al. Control-flow integrity. In *ACM CCS*, pages 340–353. ACM, 2005.

[2] P. Akritidis et al. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[3] T. M. Austin et al. Efficient detection of all pointer and array access errors. 29, 1994.

[4] T. M. Austin et al. *Efficient detection of all pointer and array access errors*, volume 29. ACM, 1994.

[5] M. Bailey et al. The blaster worm: Then and now. *IEEE Security & Privacy*, 3(4):26–31, 2005.

[6] Bluespec Inc. Bluespec System Verilog.

[7] S. Chen et al. Non-control-data attacks are realistic threats. In *USENIX Security*, August 2005.

[8] C. Cowan et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium - Volume 7*, 1998.

[9] C. Cowan et al. Pointguard tm: protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, volume 12, 2003.

[10] M. Dalton et al. Raksha: a flexible information flow architecture for software security. In *ACM SIGARCH*, volume 35, pages 482–493, 2007.

[11] T. H. Dang et al. The performance cost of shadow stacks and stack canaries. In *ACM ASIACCS*, pages 555–566. ACM, 2015.

[12] L. Davi et al. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *ACM ASIACCS*, pages 40–51. ACM, 2011.

[13] A. de Amorim et al. A verified information-flow architecture. In *ACM SIGPLAN Notices*, volume 49, pages 165–178. ACM, 2014.

[14] J. Devietti et al. Hardbound: architectural support for spatial safety of the c programming language. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 103–114. ACM, 2008.

[15] Z. Durumeric et al. The matter of heartbleed. In *IMC Conference*, pages 475–488. ACM, 2014.

[16] E. A. Feustel. Tagged architecture and the semantics of programming languages: Extensible types. In *ISCA*, pages 147–150. ACM, 1976.

[17] N. Gala et al. SHAKTI processors: An open-source hardware initiative. In *VLSID*, pages 7–8, 2016.

[18] Y. Huang. Heap overflows and double-free attacks. In *www.homes.soic.indiana.edu/yh33/Teaching/I433-2016/lec13-HeapAttacks.pdf*.

[19] T. Jim et al. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference*, June 2002.

[20] A. Kwon et al. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *ACM SIGSAC CCS*, pages 721–732. ACM, 2013.

[21] D. Moore et al. Inside the slammer worm. *IEEE Security & Privacy*, 99(4):33–39, 2003.

[22] S. Nagarakatte et al. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[23] S. Nagarakatte et al. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *ISCA*, pages 189–200, 2012.

[24] S. Nagarakatte et al. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *CGO*, page 175, 2014.

[25] G. C. Necula et al. Ccured: Type-safe retrofitting of legacy software. *ACM TOPLAS*, 27(3):477–526, 2005.

[26] T. Newsham. Format string attacks. http://forum.ouah.org/FormatString.PDF, 2000.

[27] O. Oleksenko et al. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. *USENIX Annual Technical Conference*, 2017.

[28] A. One. Smashing the stack for fun and profit. phrack 49, 1996.

[29] R. Roemer et al. Return-oriented programming: Systems, languages, and applications. *ACM TISSEC*, 15(1):2, 2012.

[30] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM CCS*, pages 552–561. ACM, 2007.

[31] H. Shacham et al. On the effectiveness of address-space randomization. In *ACM CCS*, pages 298–307. ACM, 2004.

[32] A. Sotirov et al. Bypassing browser memory protections in windows vista. *Blackhat USA*, 2008.

[33] Stack Shield. http://www.angelfire.com/sk/stackshield/. Jan 2000.

[34] Stack Smashing Protector (SSP). http://www.linuxfromscratch.org/hints/downloads/files/ssp.txt. August 2007.

[35] A. van de Ven. New security enhancements in red hat enterprise linux v. 3, update 3. *Red Hat*, 2004.

[36] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. The risc-v instruction set manual. volume 1: User-level isa, version 2.0. Technical report, DTIC Document, 2014.

[37] J. Woodruff et al. The cheri capability model: Revisiting risc in an age of risk. In *ISCA*, pages 457–468. IEEE, 2014.

[38] W. Xu et al. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *ACM SIGSAC CCS*, pages 414–425. ACM, 2015.

[39] Y. Younan et al. Paricheck: an efficient pointer arithmetic checker for c programs. In *ASIACCS*, pages 145–156. ACM, 2010.

View publication stats