SHAKTI-MS : A RISC-V Processor for Memory Safety in C

Sourav Das
Indian Institute of Technology, Madras
sourav.iniesta13@gmail.com

R Harikrishnan Unnithan
Birla Institute of Technology and Science Pilani, Goa Campus
f20140181g@alumni.bits-pilani.ac.in

Arjun Menon
Indian Institute of Technology, Madras
c.arjunmenon@gmail.com

Chester Rebeiro
Indian Institute of Technology, Madras
chester@cse.iitm.ac.in

Kamakoti Veezhinathan
Indian Institute of Technology, Madras
kama@cse.iitm.ac.in

ABSTRACT

In this era of IoT devices, security is very often traded off for smaller device footprint and low power consumption. Considering the exponentially growing security threats of IoT and cyber-physical systems, it is important that these devices have built-in features that enhance security. In this paper, we present Shakti-MS, a lightweight RISC-V processor with built-in support for both temporal and spatial memory protection. At run time, Shakti-MS can detect and stymie memory misuse in C and C++ programs, with minimum runtime overheads. The solution uses a novel implementation of fat-pointers, those associate capabilities with every pointer. Our proposal is to use stack-based cookies for crafting fat-pointers instead of having object-based identifiers. We store the fat-pointer on the stack, which eliminates the use of shadow memory space, or any table to store the pointer metadata. This reduces the storage overheads by a great extent. The cookie also helps to preserve control flow of the program by ensuring that the return address never gets modified by vulnerabilities like buffer overflows. Shakti-MS introduces new instructions in the microprocessor hardware, and also a modified compiler that automatically inserts these new instructions to enable memory protection. This co-design approach is intended to reduce runtime and area overheads, and also provides an end-to-end solution. The hardware has an area overhead of 700 LUTs on a Xilinx *xcvu095-ffva2104-2-e* FPGA and 4100 cells on an open 55nm technology node. The clock frequency of the processor is not affected by the security extensions, while there is a marginal increase in the code size by 11% with an average runtime overhead of 13%.

CCS CONCEPTS

• **Security and Privacy → Hardware and Compiler security implementations** ; • **Computer systems organization → Embedded systems**; **Reduced Instruction set architecture**.

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).
xxx, 2019, xxx
© 2019 Copyright held by the owner/author(s).
ACM ISBN xxx-xx-xxx/xx/xx.
https://doi.org/x.xxx/xxx

KEYWORDS

Secure Processor Architecture, Buffer Overflows, Memory Safety, Use-after-free, Spatial Attacks, Temporal Attacks, Dangling Pointers

ACM Reference Format:
Sourav Das, R Harikrishnan Unnithan, Arjun Menon, Chester Rebeiro, and Kamakoti Veezhinathan. xxx. SHAKTI-MS : A RISC-V Processor for Memory Safety in C . In *Proceedings of xxx*. ACM, xxx, xx, xxx, Article x, 11 pages. https://doi.org/x.xxx/xxx

1 INTRODUCTION

With the advent of IoT, there has been a rapid increase in the use of low-power embedded devices. These devices are deployed in wide and diverse applications that are connected to the Internet. While these devices becoming more pervasive, large scale attacks involving compromised embedded devices such as the Mirai botnet [21] are becoming commonplace. In the absence of robust secure environments, vulnerabilities introduced in these devices due to programming flaws can allow attackers to take control of systems with ease.

Several of these vulnerabilities occur due to illegal use of memory accesses. Today, these memory access vulnerabilities rank among the top 25 vulnerabilities in system software [24]. Vulnerabilities like buffer overflows [34], use-after-free(UAF) [36, 43], and double-free [16] are some of the major security threats. These vulnerabilities still persist due to predominant use of C and C++ programming languages due to the fact that these programming languages have features like explicit pointer manipulations, flexible type casting constructs and ease in interfacing with the hardware. These features make them favorable for the development of operating systems, virtual machine monitors, embedded systems, and database management software. However, these features come with the risk of illegal memory access and have led to many attacks in the past. Rewriting all existing code in memory safe languages is not feasible and hence we are left with the difficult task of retrofitting security into existing systems.

There have been many studies relating to spatial and temporal attacks due to illegal memory uses [1–3, 5–8, 25, 26, 28–31, 35] and many have proposed methods to prevent one or both of these attacks. Some of the approaches focus only on software solutions [1–3, 5, 8, 28–31], while others rely on support from the hardware to enforce memory safety [6, 7, 25, 26, 35]. Many of the existing software solutions either fail to provide complete temporal and

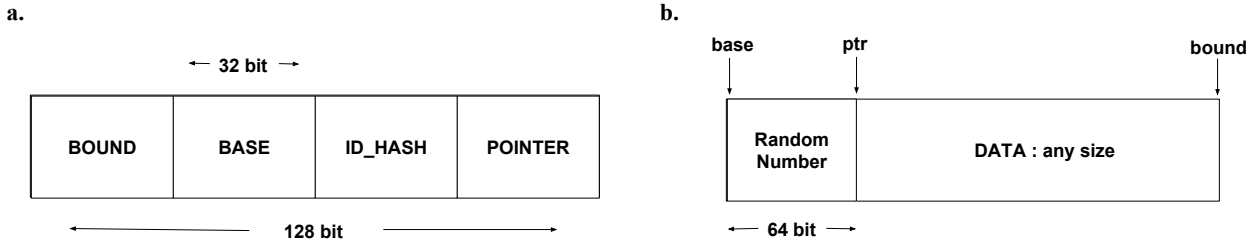

Figure 1: a. Structure of metadata in a fat-pointer.
b. Storage of metadata along with the malloc'd region

spatial safety or they incur too high run time overheads. Pure software solutions like [28] and [29] can be combined to tackle most kinds of spatial and temporal attacks, but this approach leads to high code size and runtime overheads of around 116% [29]. On the other hand, hardware solutions like [23, 25] reduce the run time overhead at the cost of hardware complexity. Although [23] enhances a RISC-V processor to efficiently implement memory checks, the software support required for [23] is extremely complex. Watchdog [25], is a compiler plus hardware solution for memory safety. It uses a shadow memory space to maintain the metadata used for memory checks. This shadow memory can result in considerable memory overheads. For complete spatial and temporal safety, 56% of the system memory would be inaccessible [25]. Gandalf [18], also has a hardware plus software solution without much extra hardware complexity and minimal compiler modifications. However, it does not provide temporal safety.

In this paper, we introduce `Shakti-MS`, a RISC-V processor providing both spatial and temporal memory safety. It requires light-weight modifications in the compiler to insert certain instructions that enable the hardware to perform the required memory checks at runtime. Further, unlike [25], we are not using any separate shadow memory space and unlike [23], there are no additional tables or tag bits that are required in the processor to store pointer metadata. These features reduce the hardware complexities and storage overhead to a great extent. Another significant benefit of our approach is that the hardware is fully compliant to the RISC-V spec. Any binary compiled with an unmodified compiler toolchain can still run on the modified processor and can coexist with protected programs. One program itself may have protected and non-protected sections by selectively building static and dynamic libraries with protection enabled. In our approach to prevent spatial and temporal attacks, each derived data type object (pointers, arrays, structures) is transformed into a fat-pointer as shown in Figure 1.a. In addition to the memory pointer, the fat-pointer also contains the *base*, *bound* and the *id_hash* fields. The *base* and *bound* are used to ensure spatial safety, whereas the *id_hash* field is used to ensure temporal safety. To protect against illegal memory operations in stacks, each stack frame is associated with a cookie to help craft fat-pointers for objects within the current stack frame. The idea of having a stack based cookie instead of an object based identifier helps reduce the storage overheads, as temporal metadata is associated with stack frames rather than individual variables. The cookie also helps to preserve the control flow of the program by ensuring that the return address never gets modified. Moreover, the cookie not only helps to prevent temporal attacks in stacks but also serves as the lower bound to prevent any overflows beyond this region. To provide memory safety in heaps, each allocated region is associated with a unique 64-bit value to craft fat-pointers as shown in Figure 1.b. The base address of pointer referencing to this malloc region would point to this unique random number. This number also acts as the cookie for this allocated region. All pointers pointing to the same allocated region uses the cookie value to craft the fat-pointer. Moreover when any one of these fat-pointers is freed the cookie value is randomised to ensure that all other fat-pointers pointing to the same allocated region is invalid. The storage overhead for the proposed solution is $(128 * n + 64)$ bits. In case of stacks, n indicates the number of derived data type objects in the current stack frame and for heaps, it indicates the number of aliased pointers, i.e. pointers pointing to the same allocated region of memory. We introduce two new instructions namely **"hash"** and **"val"** which are added in the RISC-V ISA to support memory safety checks. These instructions are inserted by the compiler at the desired places to ensure that all pointer accesses are validated before being performed. To achieve the compiler modifications, a transformation pass is developed in RISC-V LLVM [19] and the hardware support is developed in Bluespec-System-Verilog [4].

The rest of the paper is organized as follows. Section 2 defines some of the terminologies used in `Shakti-MS`. It also discusses the key idea that prevents temporal and spatial attacks on stacks and heaps. Section 3 elaborates the architecture and implementation of `Shakti-MS`. This section also describes the compiler modification along with the details of Micro-architectural implementation of `Shakti-MS`. Section 4 presents some of the case studies demonstrating how the compiler changes were made, and how these changes help thwart certain attacks. Section 5 reports some of the analysis like runtime overheads and code size overheads of `Shakti-MS`. Section 6 concludes the paper.

2 SHAKTI-MS : THE CRUX

This section describes the proposed solution and explains how the system is protected against various memory related attacks. Before diving into the solution, we first define the terminologies

that will be used subsequently. We will then describe how spatial and temporal attacks are prevented on stacks and heaps.

2.1 Terminologies

(1) **Stack Frame Cookie (SFC)** : It is a unique 64-bit random number that is placed on the stack frame below all the variables of the current function as shown in Figure 2. This SFC is unique for each function call and is used to provide temporal safety for all variables and objects available within the current stack frame. Moreover, the SFC is destroyed once the function goes out of scope ensuring all pointers to be invalid once the function returns.

(2) **RODataCookie (RODC)** : Just like the SFC, it is also a unique 64-bit random number, but unlike the SFC, it is placed in the `.bss` region of the program's memory. It is used to protect the read only segment of memory thereby preventing any kind of over-reads or invalid pointer accesses. The RODC is used to provide temporal safety for both global and static variables.

(3) **ID_HASH** : It is a 32-bit unsigned number computed either from the cookies of stacks or heaps. It is also one of the four fields of the fat-pointer. The value of *id_hash* is computed with the help of a new **"hash"** instruction which is described in Section 3.1.1.

(4) **BASE** : It is a 32-bit value indicating the base address of the respective cookie. It is one of the four fields of the fat-pointer which is used for computing hash values and for checking lower bounds.

(5) **BOUND** : It is a 32-bit address indicating the absolute bound of the object. It is the maximum permissible range that the fat-pointer can access.

(6) **Safe Malloc** : It is a wrapper function (similar to the malloc function) that allocates 8 more bytes than the requested size of malloc, as shown in Figure 1.b, and returns a fat-pointer corresponding to the allocated region. In this extra 8-bytes we store a unique 64-bit random number (a cookie) which helps us to protect against temporal attacks. This cookie is used to craft fat-pointers for all pointers pointing to the allocated region of memory.

(7) **Safe free** : It is a wrapper function (similar to the free function) that accepts fat-pointers instead of normal memory pointers as its input. The method *safefree* first validates the fat-pointer. On successful validation, it calls the *free* function (after converting the fat-pointer into a normal pointer and passing it as the input) that deallocates the corresponding memory region. This method also randomises the 64-bit value stored along with the allocated region of memory, so that any further reference to that region would result in an invalid memory access.

(8) **Craft** : It is a function that is used to craft fat-pointers. It accepts four 32-bit numbers i.e *base*, *bound*, *id_hash*, and the *pointer* itself and then returns a 128-bit object by creating the fat-pointer. Figure 1.a shows the structure of the fat-pointer returned by the craft function.

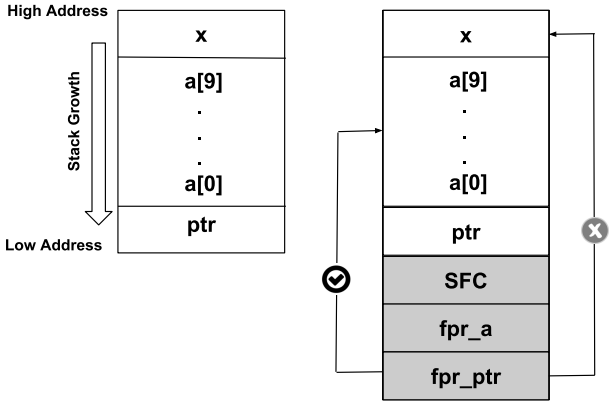

Figure 2: Stack layout with and without fat-pointers

2.2 Preventing Temporal and Spatial attacks on Stacks

Stack is often the primary target of spatial attacks primarily because overflowing local variables could potentially allow the return address to be altered, thereby changing the control flow of the program. Temporal attacks on the stack are not as prevalent as spatial attacks, though they are not unheard of [12]. A dangling pointer to the stack is a pointer of one function that is not deleted when the function returns. This pointer could now potentially be used to modify the stack of another function that later uses the same region of memory. Moreover, spatial attacks like buffer overflow have evolved over time and given rise to more sophisticated techniques like return-to-libc [38] or Return Oriented Programming (ROP) [37]. Many of these attacks have mitigations in place, such as making the stack non-executable [40] or adding stack canaries [11] to detect tampering of return address. One of the promising and most widely used solution is Address Space Layout Randomization (ASLR) [39]. Although ASLR has proven to be the most successful solutions for preventing ROP, it does not address the underlying issue of buffer overflow. There have also been attacks that bypass ASLR [22]. Another, less prevalent solution is using fat-pointers where every pointer is associated with some metadata that is used to prevent various memory corruption attacks. In our proposed solution, we use the concept of fat-pointers but differ in the implementation with respect to other existing fat-pointer solutions.

2.2.1 Preventing Spatial Attacks. : To prevent spatial attacks on stack, each derived data type object is associated with a base and bound. The bound represents the maximum accessible range the pointer to an object can access, whereas the base represents the base address of the SFC. Although the base here is not a strict lower bound for the object but it prevents all overflows provided that there are no pointer decrements. Even if there are pointer decrements it can never overflow beyond the SFC. Moreover, a slighter loose lower bound is chosen because it allows the same SFC to be used for temporal checks. To understand the concept to protect against spatial attacks let us take a look at the example below:

```
int x, a[10];
int *ptr = a;
```

```
x = *(ptr + 5);
x = *(ptr + 10); // spatial check violation
```

Figure 2, shows the stack frame with and without the fat-pointers. The shaded region represents the metadata for the stack frame. The SFC represents the "Stack Frame Cookie" which is used to protect the current stack frame and craft fat-pointers for the objects within the current stack frame. The objects placed below the SFC, namely *fpr_a* and *fpr_ptr* represents the fat-pointers for their respective objects *a* and *ptr*. These fat-pointers are placed below the SFC to ensure that there is no tampering of the metadata due to pointer decrements. Moreover, when the pointer is assigned to point to the array in the stack, the metadata of the array is copied to the metadata of the pointer. Also, every load and store instructions to this object are preceded by a validity check to ensure memory safety. So in the above example, the pointer *ptr* can only access (ptr + 5) and would fail to access (ptr + 10).

2.2.2 Preventing Temporal Attacks. : Temporal attacks on stack occur when a pointer to a local variable of a function is not deleted after the function returns, allowing it to overwrite the stack of any other function that occupies the region of the stack used by the previous function. Consider the following code snippet that demonstrates how a temporal attack can occur on stacks and how the proposed solution mitigates such attacks:

```
int* q;
void foo() {
    int a;
    q = &a;
}
int main() {
    foo();
    ... = *q; //temporal check violation
}
```

As mentioned before, each function has its own unique SFC which is used to derive the *id_hash* of each pointer in the current stack frame. Moreover, every pointer to objects in stack have its own metadata associated with it. Relating this fact to the above example, there are two functions which have their own unique SFC. When the the global pointer *q* points to the variable *a* in function *foo*, it has its *id_hash* derived from the SFC of function *foo*. As soon as *foo* goes out of scope, SFC of that function is randomised. Therefore when *q* gets dereferenced in *main* after returning from *foo*, it results in a validation error.

2.3 Preventing Temporal and Spatial attacks on Heaps

The heap is dynamically allocated region in memory that the program uses at runtime to typically store program data. A heap overflow or an overrun is a type of buffer overflow that occurs in this heap data area. Exploitations are performed by corrupting the heap data in specific ways to cause programs to overwrite internal structures such as a linked list or malloc metadata. Moreover, overflowing buffers in heap can also change pointers that point to important data. Attacks like use-after-free [43] and double-free [16] are quite common in heaps which have led to critical system failures.

2.3.1 Preventing Temporal Attacks. There have been many solutions proposed to mitigate temporal attacks [2, 3, 9, 10, 15, 29, 31, 32, 35, 41, 42]. They can be classified into two categories, namely, "location based" and "identifier based". Location based [15, 17, 31, 41] approaches use an extra data structure such as a tree or it uses a hashtable/trie-based implementation of shadow memory space to keep track of the allocated and deallocated memory regions. This approach prevents most dangling pointers but fails to protect against stale pointers which points to reallocated memory, as it uses the object's address to determine if the pointer is valid or not. In contrast, the identifier based [25, 27, 29, 35, 42] approach uses metadata associated with each pointer or a lock-and-key mechanism to prevent the exploitation of dangling pointers.

In Shakti-MS we use a lock-and-key based approach to mitigate all variants of temporal attacks. To protect against dangling pointers, double-free and other temporal attacks, all calls to malloc and free are replaced with *safemalloc* and *safefree*, which are wrapper functions that add metadata to heap-allocated objects. These are the basic function calls in C that provide low level access to memory. So protecting these functions are of prime importance. The *safemalloc* function call allocates an extra 8 bytes of memory, stores a unique 64-bit random number in it and returns a 128-bit fat-pointer having the pointer metadata. In the metadata, the *id_hash* field represents the key and the *base* represents the lock location. Every pointer pointing to the allocated region will be transformed into a fat-pointer with their respective *id_hash* and *base*. To ensure temporal safety, the following check is performed on dereferencing a pointer:

```
if ( id_hash != hash(memory[base]) )
    abort();//dangling pointers detected
```

The hash function is introduced because even if the lock location is compromised due to implementation flaws or by any other means, the hash function still remains unknown. This introduces an extra level of difficulty for the attacker to craft any arbitrary fat-pointers. Furthermore, all subsequent loads and stores on the fat-pointers are prefaced by temporal safety checks. The *safefree* method randomizes the 64-bit value stored at the start of the allocated region, which further ensures that any pointer dereference to that allocated region after being deallocated would result in a validation error. The other method which might cause a problem for dangling pointers is *realloc*. To ensure safe handling of reallocations, the *saferealloc* method replaces all *realloc* calls in the program. *Saferealloc* accepts a 128-bit fat-pointer to an object and the reallocation size as parameters. It validates the accepted fat-pointer, allocates a new region in heap, copies data over, frees the old region, and returns a valid fat-pointer for the newly allocated region.

2.3.2 Preventing Spatial Attacks. : Spatial attacks, as the name suggests, involves accessing regions of memory beyond the legitimate and intended scope of the code. These attacks are often accomplished by overflowing buffers in memory or reading beyond the limits of an object. To ensure spatial safety on the heaps, we use fat-pointer to restrict memory access of a pointer within a base and bound address. As discussed in the previous section each malloc'd region is now associated with a unique 64-bit random number and every pointer pointing to the malloc'd region has been transformed into a fat-pointer. Every pointer now has its own *base* and *bound* associated with it, where the *base* points to the start of the allocated

region and the *bound* points to the end of the allocated region, referring the absolute memory address the pointer can access. All pointer dereferences undergoes a base and bound check, ensuring spatial safety.

For better clarity on the proposed solution, consider the following sample code:

```
1. int *p,*q,*r;
2. p = malloc(10*sizeof(int));
3. q = r = p ;
4. int value = *(r+10); // spatial safety violation
5. free(p);
6. ... = *q; //temporal safety violation
```

The pseudo code equivalent of the above block after the compiler transforms:

```
1.  __int128 fpr_p,fpr_q,fpr_r;
2.  fpr_p = safemalloc(10*sizeof(int));
    //safemalloc returns a __int128 object
    //consisting of base,bound,id_hash and pointer
3.  fpr_q = fpr_r = fpr_p ;
4.1 validate (fpr_r+10); //Spatial violation detected
4.2 int value = *(fpr_r+10);
5   safefree(fpr_p);
6.1 validate fpr_q; //Temporal violation detected
6.2 ... = *fpr_q;
```

As shown above in line numbers 2 and 5, the compiler replaces every *malloc/free* calls with *safemalloc* and *safefree* wrapper functions. Moreover, the compiler also inserts validity checks before every pointer dereferences as shown in line numbers 4.1 and 6.1, ensuring both temporal and spatial safety.

3 ARCHITECTURE AND IMPLEMENTATION OF SHAKTI-MS

In the previous section, we looked for mechanisms to prevent spatial and temporal attacks on stack and heap. In this section, we look deeper into compiler and hardware instrumentation aspects of `Shakti-MS`.

In order to provide security guarantees, the given code might need to be instrumented. This can be done at the binary, compiler or the source code level. Additionally, new hardware instructions can also be added to accelerate memory safety checks. Binary level code instrumentation works to modify the source code after compilation. This approach, however, limits the flexibility of code instrumentation. For example, one cannot add a new instruction in between, without affecting the branch instructions that work on relative addresses. Source code transformation, on the other hand, doesn't provide one with enough information to apply transformations. For example, the stack organisation is not visible at the source code level.

However, compiler based instrumentation does not have any of the above mentioned drawbacks. Therefore, in our solution, we use compiler based transformations to achieve code instrumentation. Also, new hardware instructions are added to have minimal performance overheads while performing these security checks. The details of the implementation are given below.

3.1 ISA Extensions

This section describes the details of the two new instructions that are added to the RISC-V ISA in order to support memory safety.

(1) **hash** Instruction : The **hash** instruction is used to compute the *id_hash* field of the fat-pointer. The instruction receives the base address of the SFC for stacks, the base address of the RODC for global variables, or the base address of the heap-allocated region. It returns a 32-bit hash of the value stored at the memory location passed as an argument. The instruction is in the form of

```
hash rd, rs1
```

where the base address resides in *rs1* register and the computed hash value is stored in the register *rd*. The instruction computes

```
id_hash = hash(memory[base])
```

(2) **val** Instruction : The **val** instruction is used to validate the fat-pointer. It takes two arguments, the lower 64-bit of the fat-pointer and the higher 64-bit of the fat-pointer. It performs temporal and spatial validity checks on the fat-pointer. The **val** instruction is of the form

```
val rs1, rs2
```

where *rs1* represents the higher 64-bit and *rs2* represents the lower 64-bit. The **val** instruction performs the following checks:

```
if(base == NULL)                    //Check 1
    abort();
if(id_hash != hash(memory[base])    //Check 2
    abort();
if(ptr < base || ptr >= bound)      //Check 3
    abort();
```

Here, *Check 1* and *Check 2* ensure temporal safety by verifying that *id_hash* stored along with the fat-pointer and hash computed from value stored in the memory location pointed by *base* are equal. *Check 3* ensures spatial safety by verifying that every pointer access is within the base and bound. This prevents all manifestations of spatial and temporal memory attacks.

3.2 Compiler Based Instrumentation

The compiler based instrumentation needed in `Shakti-MS` is implemented using the RISC-V LLVM [19] compiler infrastructure. The LLVM toolchain converts the C-code to an intermediate representation (IR), runs certain passes on the intermediate representation (for instrumentation or optimization) and finally compiles the IR into machine code. The ability to write transformation passes that operate at the IR level provides great flexibility in terms of making changes at a logical level. To ensure spatial and temporal safety from the compiler perspective, we wrote compiler passes to analyze programs and understand the program behavior. We then wrote transformation passes that operate on the IR to add metadata to pointers and insert runtime checks.

In our solution, we have added support for two new machine instructions, namely **"hash"** and **"val"**, in RISC-V LLVM, with the help of intrinsics. These intrinsics are represented as function calls at the LLVM-IR level but will be translated into machine instructions at the assembly level. As per LLVM's documentation [20] adding a

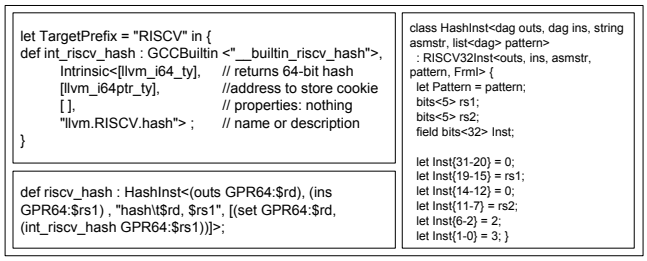

Figure 3: Code for adding "hash" intrinsic in RISC-V LLVM

new instruction directly changes the bit code format, and would require a considerable amount of effort to maintain compatibility with the previous versions. Thus, we have proceeded with adding a new intrinsic to the compiler instead of an instruction. The code for adding a new intrinsic in RISC-V LLVM is shown in Figure 3.

To explain the process of implementing the IR transformation pass, we divide it into a set of tasks (not necessarily in an order) that were performed.

(1) **Handling Global variables and global pointers** : This part of the transformation pass deals with handling global variables and global pointers which might cause a potential threat. Since these variables neither reside in the stack nor on the heap so we cannot directly craft a fat-pointer using the SFC or the metadata stored along with the malloc'd region. These variables lie in the read only section of the memory known as **.bss**. To prevent overflow or read-past-bounds attacks on global pointers, we have crafted fat-pointers using the RODC instead.
(2) **Replacing *malloc* calls and *free* calls with *safemalloc* and *safefree*** : This part of the transformation pass replaces all *malloc* calls with *safemalloc*, *free* with *safefree* and *realloc* with *saferealloc*. It also mutates the return type of the *malloc* and *realloc* functions to fat-pointers.
(3) **Adding the stack frame cookie** : This transformation pass inserts instructions to the first basic block of every function in IR. The inserted IR code generates a SFC and places it at the bottom of the stack frame every time the function is called at runtime. Once the SFC is placed on the stack, its hash value is computed using the **hash** instruction and stored in some temporary register in LLVM. This hash value is used to create fat-pointer for derived datatype objects on stack. The transform also adds code to the last basic block of the function which randomizes the SFC before the function returns. This ensures that once the function goes out of scope and returns to the calling function, any attempts to use pointers to that stack frame will raise an exception.
(4) **Handling pointer arguments/returns for function calls within the module and outside the module** : This part of the pass operates on function calls and function prototypes within the module and converts every pointer in the arguments or return values to fat-pointers. However, system calls like *scanf*, *printf* or function calls outside the module are left untouched. Any fat-pointers passed to them as arguments are first validated and collapsed into pointers. Additionally, to protect against overflows caused by special library functions like *memcpy* and *strcpy*, explicit checks are added to ensure destination buffer length is greater than source buffer length.
(5) **Crafting fat-pointers** : Since our solution is based on fat-pointers, this part of the transformation pass deals with transforming every derived data type objects on stacks to fat-pointers. The fat-pointer is created by calling the function named *craft*. Then all uses of the existing object are replaced with the newly created fat-pointer to ensure memory safety.
(6) **Transformations for various LLVM instruction** : This is the most important part of the transformation pass converting pointers to fat-pointers and handling type mismatch of all LLVM instructions. It is also responsible for adding **val** instruction before every load and store instruction. It also ensures that wherever a pointer is dereferenced, a validity check is inserted just before it. The validity checks are only inserted by the compiler in the form of a **val** instruction, but the actual check is performed by the hardware at runtime.
(7) **Warnings for Pointer Decrements** : This is an analysis pass used for identifying any decrement operation performed on pointers within the program. As stated earlier, pointer decrements on stacks might cause illegal access to other objects within the current stack frame, below the base element of the pointer. Thus, this pass of the compiler is responsible for throwing a warning whenever a pointer decrement is encountered in the desired function or module. Currently, no automated solution has been put in to fix this problem; therefore, its the responsibility of the programmer to handle this scenario. A possible solution can be to replace the pointer decrement with a pointer plus offset for compiler-enforced safety, or manually validate the safety and suppress the warning.

3.3 Micro Architecture

The hardware and ISA extensions proposed for `Shakti-MS` 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 6-stage in-order Shakti C-64 design [14] as our baseline processor whose micro-architecture is shown in Figure 4. This processor was designed using Bluespec System Verilog (BSV) [4]. Following is a brief outline on the functioning of Shakti C-64:

(1) **PC Generate Stage**: This is the first stage of the pipeline. This stage is responsible for generating the value of the Program Counter (PC). The Branch Prediction Unit (BPU) sends out the value of PC and the prediction bits. If the prediction bits indicate that the branch is taken, the next PC is the value that is given out by the BPU; else, the next PC is computed as current PC + 4. This PC is then sent to the instruction cache.
(2) **Fetch Stage:** The response of the instruction cache is read in this stage. Once the instruction is received, it is then sent to the branch predictor, and also enqueued inside the IF (Instruction Fetch) - ID (Instruction Decode) Inter-Stage

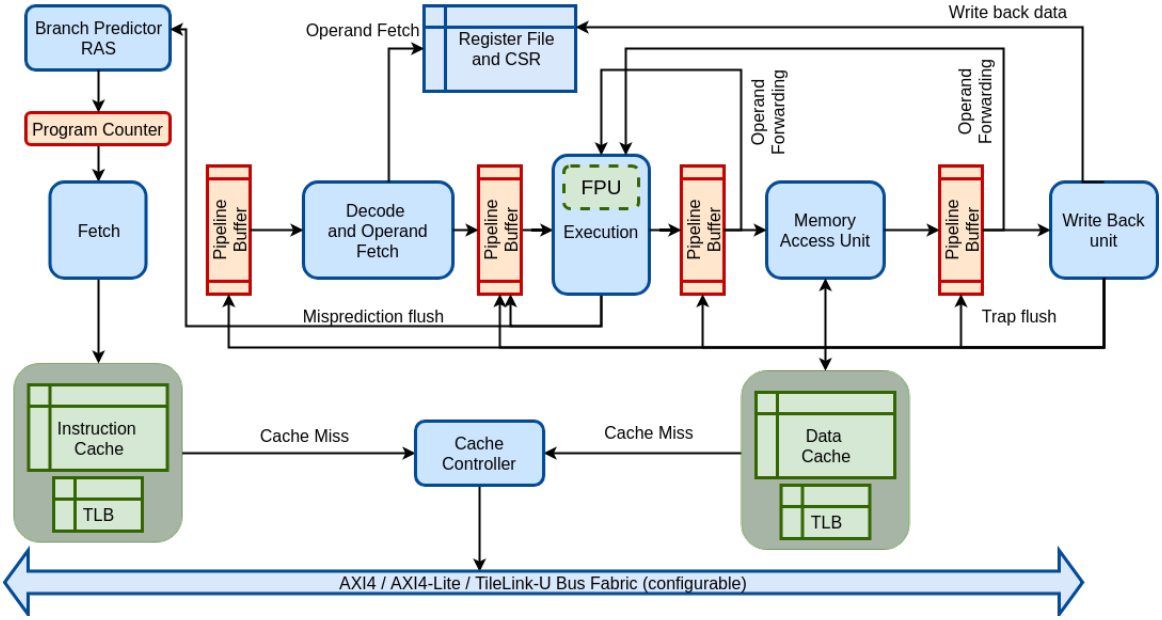

Figure 4: 6-stage pipeline processor depicting micro-architecture of Shakti-C

Buffer (ISB) provided there was no bus error or a misaligned address exception.

(3) **Decode Stage:** The instruction from the IF-ID ISB is decoded in this stage. The decoding process involves identification of the type of instruction, the source operand registers, the destination register, etc.. The results are then stored in the ID - EXE (Execute) ISB.

(4) **Execute Stage:** In this stage, the operands are fetched from the respective register file, or the operand forwarding path and the instruction is executed using the ALU (Arithmetic-Logic Unit) for all instructions except the Floating Point (FP) instructions. The FP instructions go through the dedicated FP Unit that are IEEE-754 2008 compliant. Additionally, for a branch instruction, the prediction is validated in this stage, and the result is then sent to the BPU where the prediction bits (that decide if a branch is taken or not taken) are updated accordingly. If the branch was mispredicted, then the IF-ID ISB is invalidated and the PC is set to the correct address. Also, for memory instructions, this stage simply calculates the effective memory address. The results of this stage are stored in the EXE - MEM (Memory) ISB.

(5) **Memory Stage:** In this stage, if the instruction executed is found to be a load/store then the request is sent to the data cache to perform the necessary memory/IO operation. On completion of the memory/IO access, if there was no bus error, the cache responds back with a valid data (for load instructions) or a valid acknowledgement (for store instructions). Memory operations could also return a misaligned exception. This response is stored into the MEM - WB (Writeback) ISB. However, if the instruction executed did not require any memory accesses then the result from the EXE-MEM ISB is simply buffered into the MEM-WB ISB.

(6) **Writeback Stage:** This stage is responsible for writing the results back to the register file if no exception was generated. Also, the result is forwarded via the operand forwarding path. In case of an exception, a complete pipeline flush is initiated, and the processor jumps to the exception handler routine. Also, for instructions like store and branch, no operations are performed in this stage.

As far as the hardware implementation of the two new instructions are concerned, the PC Generate, Fetch and Decode stages for both of them work similar to that of any other arithmetic instruction. Actions performed in the remaining stages are described below:

(1) **Hash instruction**

- **Execute stage:** In the execute stage, the **hash** instruction is treated similar to that of a load instruction. Here the effective address is resolved by retrieving the address from the *rs1* register. Moreover, an extra piece of information is passed onto the memory stage to distinguish between normal loads and the hash instruction.
- **Memory stage:** In this stage, memory access is performed and the pipeline stalls until a response is obtained from the memory subsystem. Once a response is obtained it is checked for exceptions. If the response was a valid one, then the hash of the read value is computed and written onto MEM-WB ISB. Additionally, the hash value computed is forwarded via the operand forwarding path.
- **Writeback stage:** The writeback stage of a **hash** instruction is similar to that of a load instruction. The results are forwarded via the operand forwarding path and also written into the register file, provided there was no exception generated.

(2) **Val instruction**

- **Execute stage:** In this stage, two actions are performed by the processor. First, the effective address is resolved i.e. the address of the base is extracted from the operands. To be more precise the address is the lower 32-bits of *rs1* register. This address will be further used in the memory stage to issue a load request to get the cookie. The second action is to compare the value of the pointer with the values of base and bounds that are present in the fat-pointer. If the pointer lies within its permissible limits, no exception is raised, else an exception bit is set and the result is forwarded to the subsequent stages.
- **Memory stage:** The **val** instruction in this stage basically performs three operations provided that the exception bit was not set in the execute stage. Initially, it issues a load request to the effective address that was computed in the execute stage. Once the response is obtained, necessary checks for exceptions are performed, and on valid response only, the hash of the returned value is computed. Finally, the computed hash is compared with the *id_hash* stored along with the fat-pointer (*id_hash* stored in the upper 32-bits of *rs2*). If these values match, then the load is treated to be valid, else an exception bit is set to indicate invalid memory accesses by the pointer. The results obtained in this stage are then passed onto the writeback stage.
- **Writeback stage:** This stage reads the data from the previous stage and checks if the exception bit is set. If so, then an *Invalid_Pointer* exception is raised; else no operation is performed thereby indicating that the subsequent load or store instruction is indeed a valid access.

4 CASE STUDY

This section provides a sketch of the generated LLVM-IR code (by Clang) of different parts of a simple C-program, and also how the transformation pass modifies the IR. Given below are some of the examples of the transformation pass.

(1) **Handling the SFC** : Given below is the LLVM IR code to insert the SFC and collapse it just before the function exits. The keyword **"alloca"**is used to allocate a memory on the stack and all variables with '%' sign represent a temporary register. LLVM uses the concept of static single assignment and has infinite number of registers for computation.

```
;insert this at the end of all
;alloca calls in a function
%stack_cookie = alloca i64
%2 = call i64 @random64()
store i64 %2, i64* %stack_cookie
;body of the function call
...
;insert this at the end of the function
%4 = call i64 @random64()
store i64 %4, i64* %stack_cookie
%stack_cookie_burn = call
i64 @llvm.RISCV.hash(i64* %stack_cookie)
```

Here *@llvm.RISCV.hash* represents the intrinsic call to our function hash.

(2) **Crafting fat-pointers** : Crafting a fat-pointer is done by calling a function named *craft* with four parameters namely *base*, *bound*, *id_hash* and the *pointer* itself. The *craft* function is a few lines of assembly code inserted during code lowering. The call to *craft* function below is used to create a fat-pointer to a character array of size 10.

```
...
%stack_cookie_32 = ptrtoint i64* %stack_cookie
to i32
...
%stack_hash = trunc i64 %stack_hash_long
to i32
%2 = alloca [10 x i8], align 1
%pti1 = ptrtoint [10 x i8]* %2 to i32
%absolute_bnd2 = add i32 %pti1, 10
%fpr3 = call i128 @craft(i32 %pti1,
i32 %stack_cookie_32, i32 %absolute_bnd2,
i32 %stack_hash)
```

(3) **Validating Loads and Stores** : As stated earlier every loads and stores are prefaced by validity checks, so let us take a single line of C code and see how the following code gets transformed.

```
a[5] = *(ptr+3);
```

Here *a* is an array of size 10 and *ptr* is a pointer pointing to the array. The code below is the LLVM IR representation of the said line:

```
%2 = alloca [10 x i8], align 1
%3 = alloca i8*, align 8
%6 = load i8*, i8** %3, align 8
%7 = getelementptr inbounds i8, i8* %6, i64 3
%8 = load i8, i8* %7, align 1

%9 = getelementptr inbounds [10 x i8],
[10 x i8]* %2, i64 0, i64 5
store i8 %8, i8* %9, align 1
```

Assuming that fat-pointers exist for the variables *a* and *ptr*, the different instructions get modified as follows :

```
...
;Fat-pointer to a
%fpr3 = call i128 @craft(...)
...
;Fat-pointer to ptr
%fpr6 = call i128 @craft(...)

%fpr_low13 = trunc i128 %fpr6 to i64
%fpr_hi_big14 = lshr i128 %fpr6, 64
%fpr_hi15 = trunc i128 %fpr_hi_big14 to i64
call void @llvm.RISCV.validate(i64 %fpr_hi15,
i64 %fpr_low13)
%ptr32_16 = and i64 %fpr_low13, 4294967295
%ptrl = inttoptr i64 %ptr32_16 to i128*
%fpld = load i128, i128* %ptrl, align 8

%zextarrayidx17 = zext i32 3 to i128
%arrayidx18 = add i128 %fpld, %zextarrayidx17
;validate arrayidx18
...
%ptrl23 = inttoptr i64 %ptr32_22 to i8*
%5 = load i8, i8* %ptrl23, align 1
%zextarrayidx24 = zext i32 5 to i128
%arrayidx25 = add i128 %fpr3,
;validate arrayidx25
...
%ptrs30 = inttoptr i64 %ptr32_29 to i8*
store i8 %5, i8* %ptrs30, align 1
```

The above code also shows that all the *getelementptr* instructions have been transformed to offsets and added with the

fat-pointers to point to the desired location of memory.

(4) **Handling external function calls like *strcpy*** : To handle external function calls like *strcpy*, explicit checks for length of the destination and source buffer need to be performed. This is because we do not have any control of the external function and these can cause overflows of buffer. Let us look at the sample code below :

```
char a[10],b[10];
...
strcpy(a,b);
```

The LLVM representation of the following code is given below:

```
...
%5 = getelementptr inbounds [10 x i8],
[10 x i8]* %2, i32 0, i32 0
%6 = getelementptr inbounds [10 x i8],
[10 x i8]* %3, i32 0, i32 0
%7 = call i8* @strcpy(i8* %5, i8* %6)
```

The modified LLVM-IR code is :

```
...
%zextarrayidx = zext i32 0 to i128
;fpr3 is the fat pointer to array "a"
%arrayidx = add i128 %fpr3, %zextarrayidx
%zextarrayidx8 = zext i32 0 to i128
;fpr6 is the fat pointer to array "b"
%arrayidx9 = add i128 %fpr6, %zextarrayidx8
%fpr_low10 = trunc i128 %arrayidx to i32
%ptrc11 = inttoptr i32 %fpr_low10 to i8*
%fpr_low12 = trunc i128 %arrayidx9 to i32
%ptrc13 = inttoptr i32 %fpr_low12 to i8*
%source_len = call i64 @strlen(i8* %ptrc13)
%check_len = icmp ule i64 10, %source_len
br i1 %check_len, label %5, label %7
...
```

In the above modified code, if the condition of the branch instruction is true, then the code jumps to a basic block and finally exits; else it executes normally
.

(5) **Malloc** : In LLVM IR the malloc call would get transformed into a *safemalloc* call. Let us look at a simple example of how the original IR looks and the modified IR is generated.

```
char *q = malloc(10);
```

The corresponding LLVM IR code would look something like this:

```
%1 = alloca i8*, align 8
%2 = call i8* @malloc(i64 zeroext 10)
store i8* %2, i8** %1, align 8
```

where %1 refers to the allocation of variable *q*. *Malloc* allocates 10 bytes of memory, and assigns it to *q*. The modified IR code is given below :

```
%1 = alloca i128, align 8
...
%fpr = call i128 @craft(i32 %pti,
i32 %stack_cookie_32, i32 %absolute_bnd,
i32 %stack_hash)
%3 = call i128 @safemalloc(i64 zeroext 10)
;validate and store
...
call void @llvm.RISCV.validate(i64
%fpr_hi, i64 %fpr_low)
...
store i128 %3, i128* %ptrs, align 8
```

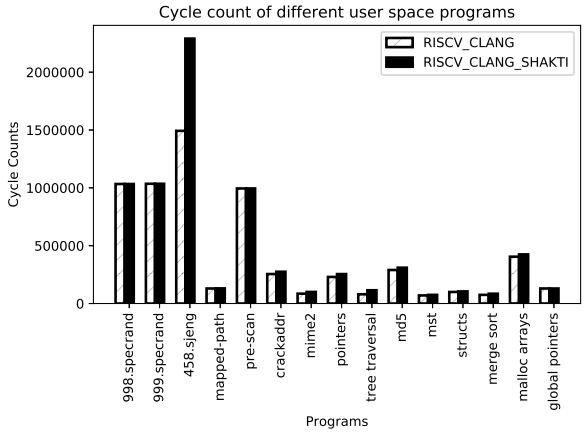

Figure 5: Graph demonstrating cycle count overheads of different programs.

(6) **Free** : Similar to the *malloc* call, a *free* call also gets modified into *safefree* and it now accepts a 128-bit fat-pointer instead of a normal pointer. Assuming that a variable *q* points to an already malloc'd memory region, a call to *free(q)* is implemented using the below LLVM instructions:

```
%3 = load i8*, i8** %1, align 8
call void @free(i8* %3)
```

where %1 represents the same variable shown in the malloc code above. The modified LLVM code is:

```
;validate and load
...
call void @llvm.RISCV.validate(i64 %fpr_hi3,
i64 %fpr_low1)
...
%fpld = load i128, i128* %ptrl, align 8
call void @safefree(i128 %fpld)
```

5 RESULTS

Shakti-MS has two implementation aspects, namely, hardware design and compiler transformations. Hardware additions cause an increase in the area of the chip, and also may increase the critical path length. The compiler transformations, on the other hand, may cause an increase in the code size and runtime overheads. This section discusses the overheads in terms of all these aspects, and also quantifies the effectiveness of the proposed solution.

5.1 Runtime Overheads :

To calculate the runtime overheads we have used some of the SPEC benchmarks that had successfully compiled using RISC-V LLVM toolchain. We also used some of the buffer overflow benchmarks given in SARD-dataset-88 [13], and some commonly used programs

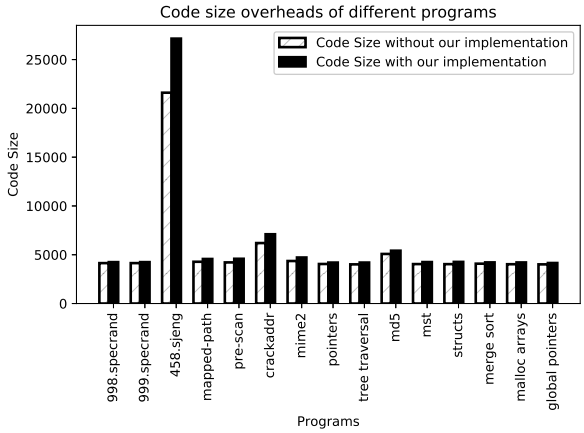

Figure 6: Graph demonstrating overheads of different programs with respect to code size.

consisting of intensive pointer operations/arithmetic to estimate the runtime overheads. The average runtime overhead is approximately 13%.

Figure 5 shows the cycle count overhead for some of the benchmarks and other programs. The white bar in the graph indicates cycle counts for execution if the programs are compiled with vanilla RISC-V CLANG, while the black bar indicates the cycle count if the programs are compiled with the modified compiler toolchain.

5.2 Overheads in the Code:

To estimate overheads in terms of code size, the *hexdump* of the same set of programs were compared with and without the LLVM transformation pass.

Figure 6 shows the code size of different programs. The white bar indicates the code size of the program without the transformation pass applied, whereas the black bar indicates the code size with the transformation pass applied. The average increase in code size is about 11%.

5.3 Hardware Overheads :

The modified microprocessor was synthesized on UMCIP's open 55nm technology node, and also on a Virtex Ultrascale FPGA (part number *xcvu0095-ffva2104-2-e*) using Xilinx Vivado 2016.1. The RAMs of caches were treated as a black box for the ASIC synthesis. Shakti-MS has an overhead of 4100 cells on ASIC, and 700 LUTs on the FPGA. The critical path, which is in the execute stage, did not change as the base and bounds check, and hash computation are done in parallel with the existing circuit in that stage. Also, the extra logic in the memory stage does not fall on the critical path.

5.4 Effectiveness :

To check the effectiveness of bounds checking and use after free attacks, we have used the SARD-dataset-81 and SARD-dataset-89 downloaded from SAMATE-NIST [13] website. We also developed our own test cases for more obscure memory corruption attacks. Our test cases were developed to target the different uses of dangling pointers for temporal safety checks, whereas the SARD-dataset was used for spatial safety checks. The SARD dataset has around 1100 programs consisting of both correct and vulnerable ones. Our solution was able to detect all the vulnerabilities it was designed to address. The issues relating to false negatives were due to multi-threaded programs and nested sub-object protection. However, issues relating to sub-objects are handled to ensure that the overflow is never beyond the object's scope. Since these are very small tests that are written just to check for the effectiveness of a solution, these have not been included in Figure 5. Nevertheless, the runtime overheads that were observed for these programs were negligible.

Table 1: Comparative Study with Existing Works

	Safety Check		Instrumentation Methods		Metadata Size	Performance Overheads	
	Spatial	Temporal	Hardware	Compiler		Hardware	Software
[33]	✓	×	×	✓	128*n	NA	NA
[27]	✓	✓	×	✓	256*n + 64	NA	29%
[25]	✓	✓	✓	✓	256*n + 64	NA	25%
[23]	✓	✓	✓	×	64*n + 128	0%	NA
[7]	✓	×	✓	×	128*n	NA	10%
Shakti-MS	✓	✓	✓	✓	128*n	0%	13%

6 CONCLUSION

In this paper, we propose `Shakti-MS`, a RISC-V processor supporting both spatial and temporal memory safety. It is a light-weight co-design approach with the compiler responsible for inserting new instructions that perform memory checks and the hardware responsible for executing them. Table 1 shows a comparative study of the runtime overhead of the proposed solution with the existing works. The low runtime overhead is achieved due to the fact that the work is being divided between the compiler and the hardware. The major contribution of the paper lies in the fact that we are using stack based cookies instead of using object based id's. The proposed implementation of fat-pointer prevents both spatial and temporal attacks on stacks and heaps with minimal storage and runtime overheads. Another major advantage of `Shakti-MS` allows existing RISC-V software and binaries to be run unmodified. This means that any program compiled with an unmodified compiler toolchain can still run on the modified processor and co-exist with the protected programs.

Although we see that `Shakti-MS` works well for protecting against both spatial and temporal attacks, but observing its effectiveness in case of sub-object protection and multi-threading environment would be an interesting work. Moreover, since our code transformation relies on the compiler to insert instructions, different optimisation passes can be applied before and after our own transformation pass. One example would be to run a pass and figure out statically as to which pointers need to be transformed into fat-pointers, and only transform those pointers to have minimal runtime and code size overheads.

REFERENCES

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *18th USENIX Security Symposium, Montreal, Canada, August*

10-14, 2009, Proceedings. 51–66. http://www.usenix.org/events/sec09/tech/full_papers/akritidis.pdf

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*. 290–301. https://doi.org/10.1145/178243.178446

[3] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. 158–168. https://doi.org/10.1145/1133981.1134000

[4] Bluespec Inc. 2003. Bluespec System Verilog. https://bluespec.com.

[5] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. 147–160. http://www.usenix.org/events/osdi06/tech/castro.html

[6] Weihaw Chuang, Satish Narayanasamy, and Brad Calder. 2007. Accelerating Meta Data Checks for Software Correctness and Security. *J. Instruction-Level Parallelism* 9 (2007). http://www.jilp.org/vol9/v9paper10.pdf

[7] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*. 103–114. https://doi.org/10.1145/1346281.1346295

[8] Dinakar Dhurjati and Vikram S. Adve. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. 162–171. https://doi.org/10.1145/1134285.1134309

[9] Dinakar Dhurjati and Vikram S. Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *2006 International Conference on Dependable Systems and Networks (DSN 2006), 25-28 June 2006, Philadelphia, Pennsylvania, USA, Proceedings*. 269–280. https://doi.org/10.1109/DSN.2006.31

[10] Dinakar Dhurjati, Sumant Kowshik, Vikram S. Adve, and Chris Lattner. 2003. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03). San Diego, California, USA, June 11-13, 2003*. 69–80. https://doi.org/10.1145/780732.780743

[11] C. Cowan et al. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.

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

[13] NIST Juliet Test Suite for C/C++. 2010. Software Assurance Metrics and Tool Evaluation. https://samate.nist.gov/SRD/testsuite.php

[14] Neel Gala, Arjun Menon, Rahul Bodduna, G. S. Madhusudan, and V. Kamakoti. 2016. SHAKTI Processors: An Open-Source Hardware Initiative. In *29th International Conference on VLSI Design and 15th International Conference on Embedded Systems, VLSID 2016, Kolkata, India, January 4-8, 2016*. 7–8. https://doi.org/10.1109/VLSID.2016.130

[15] Reed Hastings and Bob Joyce. 1991. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*. 125–138.

[16] Y. Huang. 2016. Heap overflows and double-free attacks. http://homes.sice.indiana.edu/yh33/Teaching/I433-2016/lec13-HeapAttacks.pdf

[17] Richard W. M. Jones and Paul H. J. Kelly. 1997. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *AADEBUG*. 13–26. http://www.ep.liu.se/ecp/article.asp?issue=001&article=002

[18] Gnanambikai Krishnakumar, Patanjali SLPSK, Prasanna Karthik Vairam, Chester Rebeiro, and Kamakoti Veezhinathan. 2018. GANDALF: A Fine-Grained Hardware-Software Co-Design for Preventing Memory Attacks. *Embedded Systems Letters* 10, 3 (2018), 83–86. https://doi.org/10.1109/LES.2018.2805734

[19] LLVM 2000. The LLVM Compiler Infrastructure. https://llvm.org/

[20] LLVM Documentation 2000. Extending LLVM: Adding instructions, intrinsics, types, etc. https://llvm.org/docs/ExtendingLLVM.html

[21] Michael Bailey Matthew Bernhard Elie Bursztein Jaime Cochran Zakir Durumeric J. Alex Halderman Luca Invernizzi Michalis Kallitsis Deepak Kumar Chaz Lever Zane Ma Joshua Mason Damian Menscher Chad Seaman Nick Sullivan Kurt Thomas Yi Zhou Manos Antonakakis, Tim April. 2017. Understanding the Mirai Botnet. *26th USENIX Security Symposium - August,* (2017), 1093–1110. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[22] Hector Marco-Gisbert and Ismael Ripoll. 2014. On the Effectiveness of NX, SSP, RenewSSP, and ASLR against Stack Buffer Overflows. In *2014 IEEE 13th International Symposium on Network Computing and Applications, NCA 2014, Cambridge, MA, USA, 21-23 August, 2014*. 145–152. https://doi.org/10.1109/NCA.2014.28

[23] Arjun Menon, Subadra Murugan, Chester Rebeiro, Neel Gala, and Kamakoti Veezhinathan. 2017. Shakti-T: A RISC-V Processor with Light Weight Security Extensions. https://doi.org/10.1145/3092627.3092629

[24] SC MITRE. 2011. CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/

[25] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*. 189–200. https://doi.org/10.1109/ISCA.2012.6237017

[26] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2013. Hardware-Enforced Comprehensive Memory Safety. *IEEE Micro* 33, 3 (2013), 38–47. https://doi.org/10.1109/MM.2013.26

[27] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2014. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. 175. https://dl.acm.org/citation.cfm?id=2544147

[28] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. 245–258. https://doi.org/10.1145/1542476.1542504

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*. 31–40. https://doi.org/10.1145/1806651.1806657

[30] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (2005), 477–526. https://doi.org/10.1145/1065887.1065892

[31] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 89–100. https://doi.org/10.1145/1250734.1250746

[32] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2007. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 1–11. https://doi.org/10.1145/1250734.1250736

[33] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. *CoRR* abs/1702.00719 (2017). arXiv:1702.00719 http://arxiv.org/abs/1702.00719

[34] Aleph One. 1996. Smashing the Stack for Fun and Profit. *Phrack* 7, 49 (November 1996). http://www.phrack.com/issues.html?issue=49&id=14

[35] Harish Patil and Charles N. Fischer. 1997. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Softw., Pract. Exper.* 27, 1 (1997), 87–110. https://doi.org/10.1002/(SICI)1097-024X(199701)27:187::AID-SPE78\mskip\medmuskip3.0.CO;2-P

[36] Jonathan D. Pincus and Brandon Baker. 2004. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security & Privacy* 2, 4 (2004), 20–27. https://doi.org/10.1109/MSP.2004.36

[37] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1 (2012), 2:1–2:34. https://doi.org/10.1145/2133375.2133377

[38] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. 552–561. https://doi.org/10.1145/1315245.1315313

[39] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*. 298–307. https://doi.org/10.1145/1030083.1030124

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

[41] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. 2007. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*. 273–284. https://doi.org/10.1109/HPCA.2007.346205

[42] Wei Xu, Daniel C. DuVarney, and R. Sekar. 2004. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*. 117–126. https://doi.org/10.1145/1029894.1029913

[43] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. 414–425. https://doi.org/10.1145/2810103.2813637