JuniKhyat
ISSN: 2278-4632

(UGC Care Group I Listed Journal)
Vol-8 Issue-02 Nov 2018

A Comprehensive Hardware-Software Co-Design to Safeguard Against Memory Attacks

Salil Kumar Malla
College of Engineering Bhubaneswar

***Abstract*— A severe security flaw that has frequently been used against users is illegal memory access. In this letter, we introduce Gandalf, an OpenRISC processor hardware modification that thwarts memory-based assaults in all their forms with the help of a compiler. We assign lightweight capabilities to every program variable, which the hardware verifies at runtime. Gandalf does not require significant OS upgrades and is transparent to the user. Additionally, it achieves locality with negligible hardware overhead. We showcase these functionalities by running SPEC2006 benchmarks on a customized Linux kernel. As far as we are aware, this is the first study to present a comprehensive approach to hardware-based memory protection strategies on embedded systems.**

I. INTRODUCTION

WITH the advent of Internet of Things (IoT), the numberof low-resource embedded devices deployed is likely to increase exponentially. The security of these systems is of growing concern because of the diverse cyber-physical appli- cations that will be controlled by these devices. In the absence of robust security environments, vulnerabilities introduced due to programming flaws is highly likely to be exploited. These vulnerabilities provide a means by which an attacker can subvert program execution in order to execute a malicious payload.

Several of the vulnerabilities originate due to illegal useof memory. For instance, a program execution can be sub- verted by overflowing a buffer on the program stack to modifythe return address of a function [1]. Although memory-based vulnerabilities have been known for over 30 years, they still rank amongst the top 25 vulnerabilities in system software [2].They have been used to develop a variety of malware leading to privilege escalation, denial-of-service, network penetration, and malfunction of applications.Over the years, several countermeasures have evolved to thwart memory-based attacks. Some of these countermeasures,like [3] and [4], create sandboxes to protect sensitive datafrom leakage, while others like [5]–[8], prevent malware fromexecuting by verifying the validity of each memory access. Sandboxing mechanisms often require considerable hardware modifications and are not always suited for embedded andIoT devices. Prevention mechanisms use metadata to per- form memory access validity checks. Works such as [5]–[7], which rely solely on compiler modifications to perform valid- ity checks, often have considerable performance overheads. This has resulted in techniques that off-load the pointer validitychecking to the hardware.

Most hardware-based solutions [9], [10] associate meta-data corresponding to each pointer used in the application.The metadata specifies the `base` and `bound` addresses for a pointer (`ptr`). At runtime, each memory load or store that uses the pointer, is permitted only if ((`base` `ptr`) (`ptr` < `bound`)). This pointer centric approach, however, suffers from two major drawbacks. First, there may be multiple point-ers referencing a single memory object. Each of these pointers would require their own copy of metadata resulting in consid- erable overheads. Further, it would take significant effort to invalidate the metadata of all pointers every time the asso- ciated memory object is freed. Contemporary works like [9] and [10] have overcome these issues by maintaining the meta-data of all pointers in a separate table stored in memory. Each memory access would look up the table to obtain the baseand bound. While this approach solves the problems associ- ated with multiple pointers, the separate table would break locality thus incurring performance overheads. Further, addi- tional caching would be needed to accelerate access to these tables.

In this letter, we present Gandalf, where the base and boundmetadata is associated with every stack-based memory object rather than the pointer as shown in Fig. 1. The base andbounds define the perimeter of the object. Each load and store involving the object is first validated by the object's perime- ter and only then allowed to complete. If a load or a storefails this check (e.g., access to `ptr + 18` in Fig. 1), then an illegal instruction exception is sent to the program, thereby terminating it.

Similar to [9] and [10], Gandalf overcomes the draw- backs of naïve pointer centric approaches. However, unlike these works, we do not use an explicit table to maintain metadata, hence we do not suffer from the drawbacks asso- ciated with [9] and [10]. It may be noted that such object centric approaches have been developed in software before as shown in [6]. However, to the best of our knowledge,

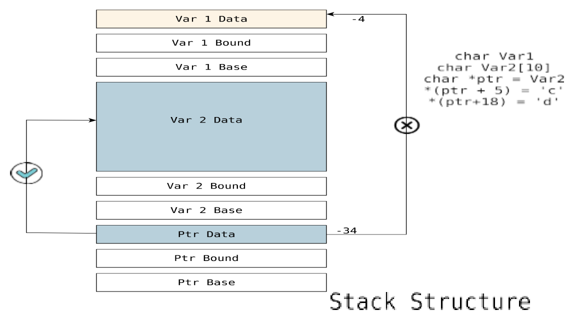

Fig. 1. Program stack layout when using Gandalf. We associate each memory object on the stack with a base and bound metadata. The pointer `ptr` can only access the buffer `var2` because its capabilities are set by the base and bound of `var2`. Any attempt to read or write to a location beyond var2 will result in an exception.

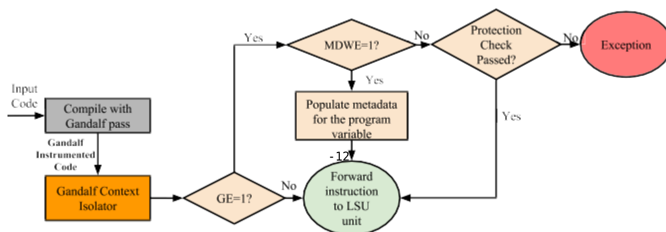

Fig. 2. Flowchart describing Gandalf scheme. Gray indicates the steps done at compiler level, orange indicates steps done at the OS level, and pink indicates the steps done by the hardware.

the metadata of the object it is dereferencing rather than the pointer's own metadata. At runtime, all load and store instruc-tions using these variables are routed through the Gandalf hardware (Fig. 2). The hardware will perform bound checks (based on the corresponding metadata) if it determines that Gandalf is enabled (i.e., GE 1) and the load/store operation is not special (i.e., MDWE 0). Special loads/stores, indi- cated by a set metadata write enable (MDWE) bit, are used to

Gandalf is the first object-based approach implemented in hardware.

We have implemented a prototype of Gandalf by extending the OpenRISC (https://openrisc.io/) architecture. We have also developed an LLVM pass for Gandalf to provide software sup-port. The setup is implemented on a field-programmable gate array (FPGA). It boots Linux and permits Gandalf enabled applications to co-exist with native Linux applications. To the best of our knowledge, this is the first work that prototypes an end-to-end system with memory protections, OS and compiler support, without resorting to any simulators.

While the object centric memory protection approach and the complete end-to-end prototype are the main contributions of this letter, Gandalf has several other features.

1) It is completely transparent to the users. The only requirement is to activate the pass at compile time with the `--gandalf` command line option.
2) In the hardware, we reuse existing special registers and instructions of the OpenRISC architecture and thus do not extend the ISA. This also results in compatibility of non-Gandalf code in the new hardware.
3) Gandalf is light weight. The impact of the hardware changes on the critical path delay is around 2% and the area overhead around 6%.

The structure of this letter is as follows. Section II gives an overview of our proposed scheme. The implementation of our solution is detailed in Section III. The security arguments supporting our scheme is presented in Section IV. The results and overheads are presented in Section V and the conclusions are presented in Section VI.

II. High Level Overview of Gandalf

We present an abstract overview of our implementation at the compiler, OS, and the hardware level. In order to enable memory protection, we enable the Gandalf option dur- ing program compilation. During the compilation every named variable in the program is augmented with metadata that spec-ifies its legal boundaries. Named variables in the program can be of three types,

namely: 1) scalar variables; 2) arrays; and
3) pointers. For bounds checking in pointers, the compiler usesill the metadata. If a protected variable tries to access a mem-ory location outside its bounds, Gandalf triggers an exception causing the program to terminate.

Programmers can choose to enable or disable Gandalf for every object file using a compile time option. Thus a single program, comprising of multiple object files, could have somefunctions that are Gandalf enabled and others that are not.The compiler ensures that function calls and memory accesseswork seamlessly between the protected and the unprotected domains. However, we only guarantee protection for pointers that are in the Gandalf enabled domains. Gandalf also permits protected and unprotected processes to simultaneously co-existin a system. This requires operating system modifications that save and restore the Gandalf context during context switches.

III. Gandalf Architecture and Implementation

In this section, we describe the implementation of the three important components of Gandalf. The first component is the compiler plugin, called *Gandalf pass*, which is invoked during compilation to infer and attach metadata (base andbounds) to program variables. The second component, called *Gandalf hardware*, is a hardware component that checks for out-of-bound memory accesses at runtime. The *Gandalf con- text isolator* is a kernel patch that permits Gandalf enabled processes to co-exist with native (non-Gandalf) ones in a system. Fig. 3 shows the Gandalf architecture.

A. Gandalf Pass

Gandalf pass is an LLVM compiler pass which, when enabled, provides the necessary information to the hardware enabling bounds checking. It also instruments code so that Gandalf and non-Gandalf functions can seamlessly interactand execute. In this section, we describe the various functions of the pass in more detail.

1) Metadata Allocation and Population: For every named variable in the program, the pass creates two additional fields for storing base and bound metadata. These fields are cre-ated at offsets 4 and 8 bytes with respect to the variable. The pass then inserts two store instructions that populate thesefields with appropriate values. A sample stack enabled with

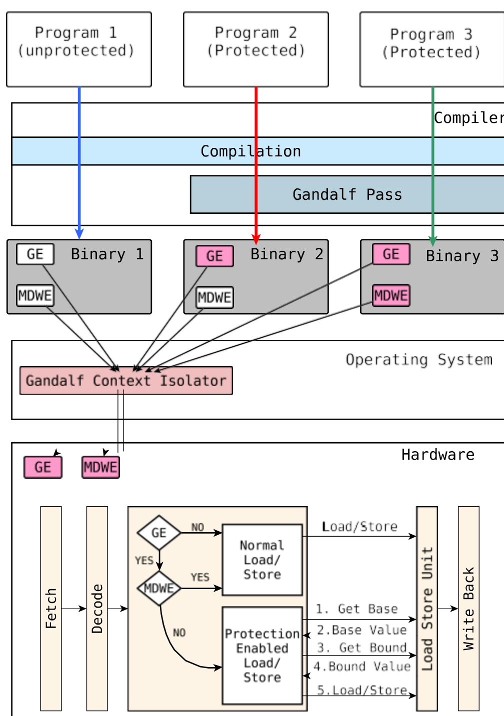

Fig. 3. Gandalf framework. Changes to the execute stage of the hardware pipeline and the Gandalf context isolator of the OS are shown. The Gandalf pass, performed during program compilation, generates executables that use Gandalf hardware. Program 1 is a legacy program while Programs 2 and 3 are Gandalf enabled. Program 3 is currently under execution in the hardware; populating the metadata.

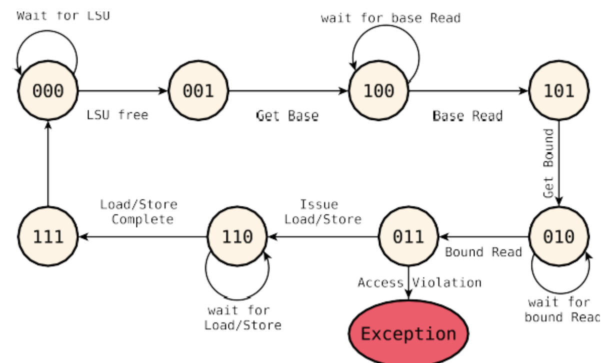

Fig. 5. Protection check state machine in the execute unit of the processor. The wait in state 000 is for the LSU to become free. The waits in 100 and010 are to let the LSU complete the previous memory read. Protection checkis performed in state 011. Exception is triggered if the check fails.

```
C code:                    Assembly code:
int arr[4];                l.addi r25, r2, -40                      } loads MDBR with
int var = *(arr+2);        l.mtspr r24, r25,<MDBR index>           } &(arr's metadata)
int *ptr = arr+1;          l.lwz r8, -24(r2)      // *(arr+2)
int var2 = *(ptr+6)        ....
...                        l.addi r25, r2, -64                      } loads MDBR with
return ret_val;            l.mtspr r24, r25, <MDBR index>          } &(ptr's metadata)
                           l.sw -56(r2), r3       // ptr = arr+1
                           ....
                           l.addi r25, r2, -40                      } loads MDBR with
                           l.mtspr r24, r25, <MDBR index>          } &(arr's metadata)
                           l.addi r3, r2, -28                       } *(ptr+6) Illegal
                           l.lwz r3, 24(r3)                         } access caught
```

Fig. 4. Instrumented assembly code generated from the Gandalf LLVMpass for a sample C code. Before every load and store instruction, the pass instruments code, shown in blue, to populate the MDBR register with the address of the corresponding metadata. `ptr`'s MDBR points to array `arr`'s metadata. Green indicates comments and red indicates the illegal access.

Gandalf protection is shown in Fig. 1. It must be noted that themetadata population is also performed through store instruc- tions. Such store instructions must skip memory protections checks since they work on metadata. Gandalf pass achieves this by enabling a bit in the system register called the MDWE right before issuing the metadata stores. The current versionof the pass provides a single base and bound for aggregate data-structures like `structs`. In the future, we plan to pro- vide fine grained base and bound metadata for elements withinthe structures.

2) Base and Bound Checking: In order to aid the hard- ware in identifying the metadata easily, we use an internal register called the metadata base register (MDBR) that storesthe metadata address. The compiler pass prepends every load and store instruction with instructions that populate the MDBR. We instrumented the compiler with the appropriate data structures to achieve this. Fig. 4 shows an example ofan instrumented code generated by the modified compiler describing the MDBR population.

3) Switching Between Gandalf and Non-Gandalf Domains: To ensure that Gandalf enabled functions seamlessly execute along with non-Gandalf functions, the compiler pass enables Gandalf (GE 1) at the start of every function of the module and disables it (GE 0) at the end of the function. This ensures that when an unprotected function calls a protected function, the protection checks are done only in the protected function and vice-versa.

B. Gandalf Hardware

When a load or a store instruction is executed by the pro- cessor, the execute stage of the pipeline performs protection checks before issuing the memory access request. To perform the checks, the execute unit uses the state machine describedin Fig. 5. It first gets the base value by issuing a memoryread to the address pointed by the MDBR. It then obtains the bound value by issuing a memory read to the address pointed by MDBR+4. The data access, i.e., the actual load or store,is allowed to proceed if and only if it is within the specified base and bounds. Otherwise, a protection violation exceptionis triggered by the hardware. In our design, we reuse the load- store unit (LSU) for performing the base and bound memory reads that are described in Fig. 5. Executing these protection checks may take multiple clock cycles to complete. We there- fore add additional control logic to stall the processor pipeline until the protection checks are completed.

We implemented Gandalf by modifying a 6-stage single issue version of the OpenRISC processor called *mor1kx cap- puccino* (https://openrisc.io/). The in-order processor has atightly coupled cache, an MMU unit, 32 general purpose reg- isters and 11 groups of special purpose registers. In order to implement the two control bits (GE and MDWE), we use the zeroth and first bits of an unused special purpose register. We use another unused special purpose register for the MDBR.

C. Gandalf Context Isolator

The Gandalf state comprises of the bits GE and MDWEand the MDBR register. We modify the OS to preserve

TABLE I
OVERHEADS OF GANDALF COMPARED WITH OTHER HARDWARE-BASED MEMORY PROTECTION SCHEMES FOR THE SPECRAND BENCHMARK FROM THE CPU2006 [11] SUITE

	Metric	**Overheads**		
		Gandalf	Shakti-T [10]	WatchdogLite [9]
Hardware	Critical Path	2.78% (0.5ns)	0%	NA
	LUT count	6.1% (666LUTs)	1914 LUTs	NA
Software	Runtime			
	Instruction Count			

80%	NA	29%
60%	NA	8 %

[1]NA implies that the numbers are not available. [2]The runtime numbers presented here are averaged over 30 runs.

the Gandalf state whenever there is a context switch. Thisis needed to permit Gandalf enabled programs to co-existwith nonprotected programs. For the Linux kernel, we mod- ified the functions `start_thread` and `_switch_to` in `arch/openrisc/kernel/process.c` to save the state of these bits on a per-process basis.

IV. Security Arguments

The base and bound metadata associated with named vari- ables in the program prevents the attacker from performing illegal memory operations. In this section, we argue that in a Gandalf enabled system, an attacker will not be able to subvertthese protection checks.

1) *Disabling Gandalf:* The attacker can disable Gandalf byresetting the GE and the MDWE bits. This is possible only if the attacker writes to the special purpose regis- ter using a malicious payload. However, the malicious payload cannot be leveraged by the attacker withoutoverflowing a buffer; which Gandalf prevents by design.
2) *Manipulating Metadata:* The attacker may also try to modify the metadata to subvert the Gandalf checks by using another pointer. However, this requires the pointerto overflow, thereby violating its base and bound checks,which Gandalf detects. Assuming that all pointers in theprogram are protected with Gandalf, access to metadata remains out of scope.
3) *Pointing to Spurious Metadata:* The attacker can attemptto mimic base and bound metadata by using some dummy data inside valid limits of a buffer. However, thisrequires a load to the MDBR register with the address ofthe dummy base. This cannot be done because it requiresan `mtspr` instruction to be executed and the attacker does not have the capability to modify the program binary.

V. Results

Gandalf requires modifications to be made in the hard-ware, OS, and compiler. We implemented the framework by extending an OpenRISC processor. The Linux kernel 4.4.0was customized to execute on the processor. The entire setup runs on a Cyclone-4 FPGA board (`EP4CE22F17C6N`) hav- ing a Terasic de0-nano FPGA. We tested the correctness of theimplementation by running several exploit codes.[1] Our imple- mentation is able to successfully prevent all the exploits from

[1]https://github.com/nekt/csaw_esc_2016/tree/master/tools/exploits

getting triggered. We were also able to run few of the SPEC 2006 benchmarks with minimal modifications.

Table I compares the overheads of our scheme with two contemporary works. It may be noted that Gandalf is the only scheme that provides end-to-end support for memory pro-tection. The work in [10] does not have OS and compiler support, while [9] provides results on a simulated hardware. Our results show that Gandalf incurs relatively less overheads both in hardware and software. The entire OpenRISC proces- sor with the Gandalf extensions utilizes 11,491 LUTs with a clock frequency of 55 MHz. The runtime overhead of soft- ware applications was found to be higher compared to [9]. However, unlike [9] we have not implemented any compiler level optimizations. There is scope to reduce the overheads of Gandalf by employing similar optimizations in our compiler pass.

VI. Conclusion

We offer a hardware–software co-design to counter memory attacks in this letter. To the best of our knowledge, this is the first effort in which memory objects, rather than pointers, are associated with base and bound metadata. We show how this could result in significantly lower overheads and lightweight, reliable memory protection systems. We have integrated the suggested methodologies into the hardware, operating system, and compiler to prototype the entire solution. When measured against other state-of-the-art efforts, the resulting solution has the lowest overheads.

Although Gandalf is effective in stopping stack-based spatial attacks, further research should be done to see how well it works against temporal attacks like use-after-free and heap-based attacks. In addition, we want to optimize the compiler in order to perhaps reduce the large runtime overheads.

References

[1] Aleph One. (1996). *Smashing the Stack for Fun and Profit*. [Online]. Available: http://phrack.org/issues/49/14.html

[2] SC MITRE. (2011). *2011 CWE/SANS Top 25 Most Dangerous SoftwareErrors*. [Online]. Available: http://cwe.mitre.org/top25/

[3] ARM. (2009). *ARM Security Technology Building a Secure System Using TrustZone Technology*. [Online]. Available: https://goo.gl/wm5b9R

[4] F. McKeen *et al.*, "Intel R software guard extensions (Intel R SGX) support for dynamic memory management inside an enclave," in *Proc. ACM HASP@ISCA*, 2016, p. 10.

[5] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safetyfor C," in *Proc. ACM PLDI*, 2009, pp. 245–258.

[6] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proc. USENIX Security Symp.*, 2009, pp. 51–66.

[7] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, “Watchdog: Hardware for safe and secure manual memory management and full memory safety,” in *Proc. IEEE ISCA*, 2012, pp. 189–200.

[8] A. Kwon, U. Dhawan, J. M. Smith, T. Knight, Jr., and A. DeHon, “Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security,” in *Proc. ACM CCS*, 2013, pp. 721–732.

[9] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, “WatchdogLite: Hardware-accelerated compiler-based pointer checking,” in *Proc. ACM CGO*, 2014, p. 175.

[10] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, “Shakti-T: A RISC-V processor with light weight security extensions,” in *Proc. ACM HASP*, 2017, p. 2.

[11] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.