ThundaTag: Disparate Domain Tagging to Enforce Benign Program Behavior

Alex Kisil, Shibo Chen
University of Michigan
{akisil,chshibo}@umich.edu

Abstract

Many attacks rely on the ability to forge pointers and overwrite architecturally sensitive addresses to gain unwarranted control in originally benign but vulnerable programs. For example, the buffer overflow attack, an attack that overwrites the return address of a function call to inject malicious code, is extremely common as a component of current deadly attacks. Current solutions only serve as minor deterrences to these attacks as they become more sophisticated. This work proposes a method of preventing the overwriting of sensitive pointers and addresses to stop control flow attacks through a rule-based system. While enforcing the benign behaviors of the program, our design only induces 3% performance overhead and 0.3% area overhead.

Introduction

Solutions to control flow attacks often take an approach of making the sensitive addresses harder to find. For example, Address Space Layout Randomization[1] originally defeated buffer overflow attacks by randomizing address bits, but was eventually thwarted as the attack was updated. Instead of making sensitive addresses harder to find, ThundaTag instead asks the question of why the user is able to overwrite such sensitive data in the first place. If the system can raise an exception when the malicious user tries to overwrite the return address of a call with their own custom pointer or immediate, the buffer overflow could simply be stopped at its core. ThundaTag sets out to prevent the overwriting of such addresses through the enforcement of a rigid and comprehensive ruleset based on disparate domain tagging of data as it propagates through the architecture.

In this work, we categorize values into 6 types, each of which has specific rules to follow in terms of what operations the user program can do on them and how to propagate under arithmetic operations. It then outlines the evaluated performance overhead and area overhead by simulation and synthesis.

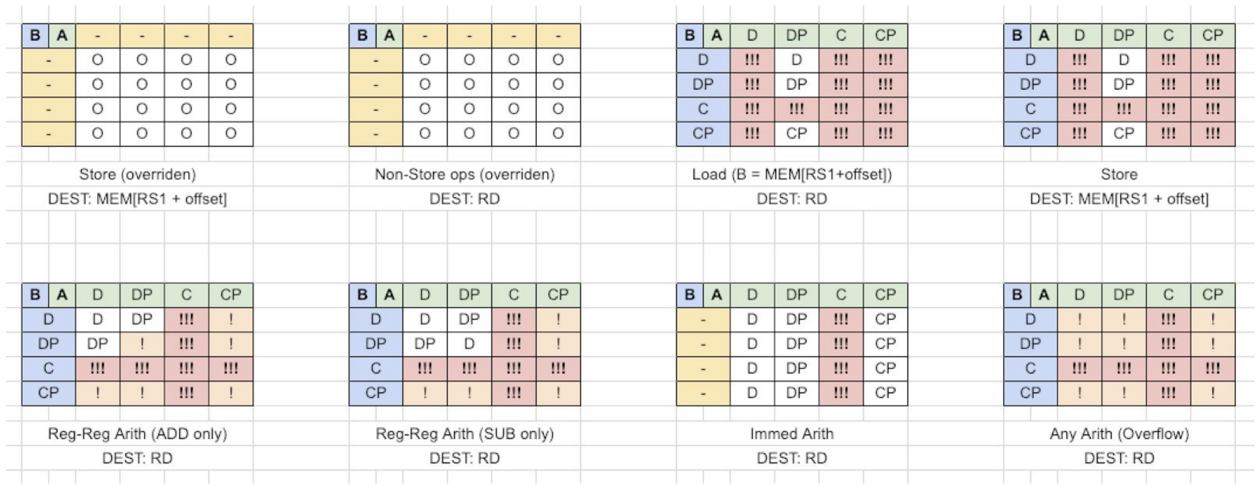
Methodology

In order to impose strict rules upon the propagation of instructions through the architecture, there must be a way of distinguishing different types of data. A system of disparate domain tagging was introduced as the fundamental core of ThundaTag. As the data is brought into the pipeline, it would be assigned one of 6 proposed tags: data, code, data pointer, code pointer, return address, and null. Data is the most basic type, in that it does not have the privilege of being returned or jumped on. Data pointers can be used to access memory, unlike data. Code pointers are the only type that can be stored in the icache and executed. They are also the only type that can be jumped/branched on (excluding returns). While conceptually, there is some overlap between these types, return addresses must be differentiated from code pointers as they should be held to certain restrictions that code pointers aren't, i.e. return pointers should never be written to, only returned on. Finally, the null type refers to any "untagged" data, such as memory that hasn't been used yet or dynamic memory that was previously freed. This type cannot be legally touched without exception.

Technical Details

The implementation of ThundaTag's designed was realized through additions to the Rocket Core in-order 5-stage pipeline. At a high level, tag checking logic was added to the ALU, and storage was allocated within the register file to accommodate the tag bits. Writeback and forwarding processes were also extended to work with tagging. 4 bits were used for each tag to allow easy alignment as well as to allow for finer grained tagging in future work.

ALU: Logic was added to the ALU module to assert the legality of operations as well as determine the tag of the operation's result. For example, the resulting tag of adding data and a data pointer should be a data pointer. A full chart of the ALU tagging logic is provided below. "!!!" indicates that an exception is thrown, whereas "!" indicates potentially suspicious behavior, and it is left up to the architect to decide how to respond.



B A	-	-	-	-
-	O	O	O	O
-	O	O	O	O
-	O	O	O	O
-	O	O	O	O

Store (overriden)
DEST: MEM[RS1 + offset]

B A	-	-	-	-
-	O	O	O	O
-	O	O	O	O
-	O	O	O	O
-	O	O	O	O

Non-Store ops (overriden)
DEST: RD

B A	D	DP	C	CP
D	!!!	D	!!!	!!!
DP	!!!	DP	!!!	!!!
C	!!!	!!!	!!!	!!!
CP	!!!	CP	!!!	!!!

Load (B = MEM[RS1+offset])
DEST: RD

B A	D	DP	C	CP
D	!!!	D	!!!	!!!
DP	!!!	DP	!!!	!!!
C	!!!	!!!	!!!	!!!
CP	!!!	CP	!!!	!!!

Store
DEST: MEM[RS1 + offset]

B A	D	DP	C	CP
D	D	DP	!!!	!
DP	DP	!	!!!	!
C	!!!	!!!	!!!	!!!
CP	!	!	!!!	!

Reg-Reg Arith (ADD only)
DEST: RD

B A	D	DP	C	CP
D	D	DP	!!!	!
DP	DP	D	!!!	!
C	!!!	!!!	!!!	!!!
CP	!	!	!!!	!

Reg-Reg Arith (SUB only)
DEST: RD

B A	D	DP	C	CP
-	D	DP	!!!	CP
-	D	DP	!!!	CP
-	D	DP	!!!	CP
-	D	DP	!!!	CP

Immed Arith
DEST: RD

B A	D	DP	C	CP
D	!	!	!!!	!
DP	!	!	!!!	!
C	!!!	!!!	!!!	!!!
CP	!	!	!!!	!

Any Arith (Overflow)
DEST: RD

Register File: To accommodate the space needed for tags in the register file, a duplicate register file was implemented solely to store tags. This register file has 4 bit cache lines so that any movement within it can mimic that of the original register file, i.e. if register 7 is overwritten in the register file, tag 7 is overwritten in the tag file.

Forwarding and Writeback: Forwarding and writeback perfectly mirror forwarding and writeback for the data that the tags correspond to. This way, tags follow their respective data through the pipeline.

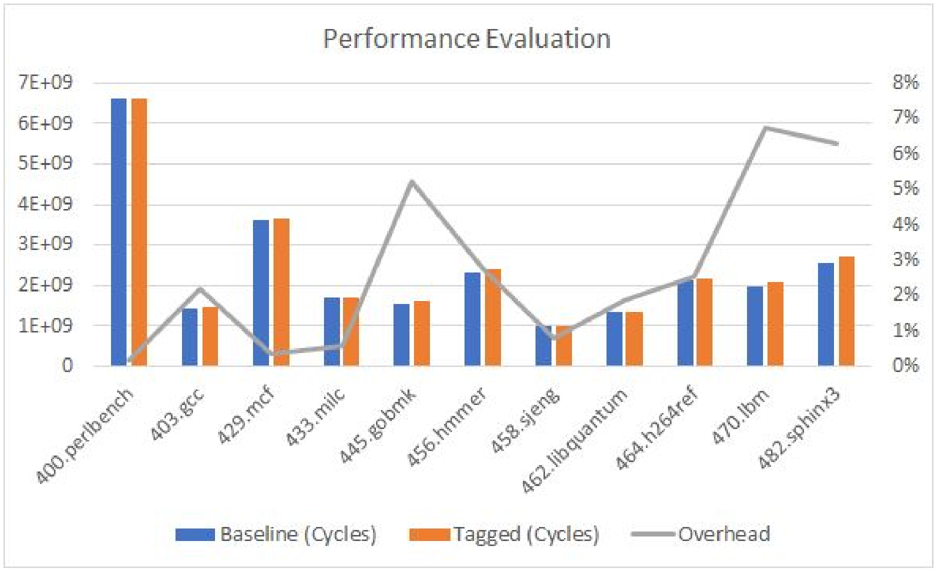
New Instructions: Two new instructions were added to the decode table in the decode phase to allow for the manual tagging and verification of data. MOVTAG casts the tag of the data in a register file to a given parameter, and CMPTAG compares the tags of two source registers.

Memory Access: In order to keep track of the tag of different values, we need to fetch tag into the pipeline along with the data associated with it. For this, we added a tag cache in the memory hierarchy. For every load operation, the system needs to bring data value from dcache and tag from the tag cache. If there is a miss in either one of the two caches, the system needs to wait until both values have been brought into the cache. For every store, the pipeline needs to make a store to tag cache at the same time. For any cache line that needs to be written back to the DRAM, the associated tag also needs to be written back from the tag cache.

Evaluation

We did evaluations on performance and area. We used gem5 to simulate performance. We then implemented a prototype based on the Rocket Core. Although it is not fully functional yet, we are still able to synthesize it and estimate the area overhead our design has.

Performance Overhead: We used gem5[3] to simulate the execution of our design. While there isn't any performance overhead introduced in the pipeline execution, the performance overhead is induced by the increasing number of memory access, either read or write, to the DRAM. In our simulation, we add one more memory access latency if there is a miss in the tag cache or the program is trying to write a piece of data back to the memory. In the performance simulation, we used an in-order core running a 1GHz and we configured the underlying hardware to have 8 kB L1 instruction cache, 32 kB L1 data cache and 2 kB tag cache. There is no L2 cache in our design. We run a set of SPEC06 benchmarks and the average performance overhead is 3%.



Area Overhead: We successfully implemented underlying logic in the Rocket Core pipeline. We used a 32-bit in-order core with 4 kB instruction cache, 16kB data cache, and no floating pointer unit. Although we have not successfully implemented logic in the caches, we put a 2 kB data array in data as the placeholder in order to get an estimation of the area overhead. After the synthesis, the area overhead is only 0.35%.

Discussion/Limitations

Due to difficulty in understanding the memory hierarchy and its interfaces within the implementation of the Rocket Core, we decided to keep tags out of the memory system for time's sake. Instead, we synthesized the project with a 2 kB larger dcache to stand in for the extension necessary for tags so that we could still get an accurate area estimate. While propagation of tags throughout the memory hierarchy is left up to future work, it is expected to have a beneficial impact on performance (as tags will reap the benefits of having a cache) but at a small detriment to area overhead. Also, a region of DRAM would be partitioned for the sole storage of tags, with a mathematical function used to calculate a datum's tag's location for load and store operations.

While we used a relatively small tag cache in our evaluation, the performance is already as low as 3%. Since the performance overhead is induced by making more memory access due to misses in tag cache, the performance can be improved if using a bigger cache or even multi-layer cache hierarchy.

Although any other utilization of the tag is out of scope for this research project, we would like to point out that it is also helpful in domain encryption and address space randomization.

Group Dynamic

The team consisted of two members of the same research lab. Because of this, we found it relatively easy to meet and discuss high-level concepts, as well as technical details in person. This was essential to our understanding of the codebase of the Rocket Core, as there is very little documentation for the project. We also kept a running shared document of any new information we learned or new questions that we had.

We utilized our physical lab space on the 2nd floor of the BBB as a meeting and workspace. In addition to our informal meetings, we also had a weekly meeting where we would present our ideas and progress to Dr. Austin and Misiker.

Conclusion

The attacker often exploits vulnerabilities by fudging data types to hijack control flow, i.e. stack overflow attack to overwrite the return address. In this project, we categorize data into different domains and set up specific rules on different domains to enforce the benign behaviors of the program. We implemented ThundaTag architectural support to power it. The design proves to have low performance overhead and area overhead.

Reference

[1] PaX Team, “PaX address space layout randomization (ASLR).” http://pax.grsecurity.net/docs/aslr.txt, 2003. [Online; accessed 6-December-2018].

[2] Asanovic, Krste, et al. "The rocket chip generator." EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17 (2016).

[3] Binkert, Nathan, et al. "The gem5 simulator." ACM SIGARCH Computer Architecture News 39.2 (2011): 1-7.