Control Flow and Pointer Integrity Enforcement in a Secure Tagged Architecture

Ravi Theja Gollapudi*, Gokturk Yuksek*, David Demicco, Matthew Cole, Gaurav Kothari, Rohit Kulkarni, Xin Zhang, Kanad Ghose, Aravind Prakash and Zerksis Umrigar
Department of Computer Science
State University of New York at Binghamton
Binghamton, New York, USA
{*rgollap1,gokturk,ddemicc1,mcole8,gkothar1,rkulka22,xzhang99,ghose,aprakash,umrigar*}*@binghamton.edu*

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179416

***Abstract*—Control flow attacks exploit software vulnerabilities to divert the flow of control into unintended paths to ultimately execute attack code. This paper explores the use of instruction and data tagging as a general means of thwarting such control flow attacks, including attacks that rely on violating pointer integrity. Using specific types of narrow-width data tags along with narrow-width instruction tags embedded within the binary facilitates the security policies required to protect against such attacks, leading to a practically viable solution. Co-locating instruction tags close to their corresponding instructions within cache lines eliminates the need for separate mechanisms for instruction tag accesses. Information gleaned from the analysis phase of a compiler is augmented and used to generate the instruction and data tags. A full-stack implementation that consists of a modified LLVM compiler, modified Linux OS support for tags and a FPGA-implemented CPU hardware prototype for enforcing CFI, data pointer and code pointer integrity is demonstrated. With a modest hardware enhancement, the execution time of benchmark applications on the prototype system is shown to be limited to low, single-digit percentages of a baseline system without tagging.**

***Index Terms*—Control Flow Integrity, Hardware security, Pointer Integrity, Security architectures, Security and privacy policies.**

1. Introduction

Recent years have seen significant growth in attacks targeting inherent vulnerabilities in software. We focus on *control-flow attacks* (CFA), which change the control flow path from what was originally intended to other paths that will ultimately execute malicious code. Examples of control flow attacks range from stack smashing [47], format string attacks [44] to the family of code reuse attacks that include return-oriented programming [7], [53], [55], jump-oriented programming [9], [13], return-to-libc programming [61], and counterfeit object oriented programming [52].

Tagged architectures have been proposed as the basis for many hardware-based security solutions against various exploits such as control flow attacks and memory corruption in general. The width of the tags in a tagged architecture determines a tradeoff between the level of protection offered and performance. Schemes that use 1-bit tags, such as HDFI [56], where one bit data tag is used to identify a single control flow related entity, incur small performance overhead. However, distinguishing between forward and backward control flow edges is not possible using a single bit. The other extreme of the tag width is found in architectures such as PUMP [26], [51] and DOVER [24], where metadata tag widths are identical to the data or instructions that they are associated with. Such wide metadata tags enable a rich and sophisticated set of security mechanisms but discourage the deployment of the tagged system due to the associated performance and storage overhead. Tag widths between these two extremes are much more useful for practical systems as they provide the right balance between the hardware-supported security mechanisms and performance. The processor architecture and system proposed in this paper shows how such a balance can be maintained to realize a practical system for ensuring the integrity of control flow and pointers by reducing performance, hardware and power overhead and yet provide a wide range of hardware-supported security mechanisms.

We introduce a Secure Tagged Architecture (**STAR**) that strikes a balance between performance and richness of tag-based security functions in order to provide protection against control flow attacks in a practical manner. STAR is a full-stack solution comprised of a FPGA (Field Programmable Gate Array) hardware prototype of the open-source Flute [10] 64-bit RISC-V ISA implementation, a modified LLVM compiler toolchain that generates the instruction and data tags, and a modified Linux kernel providing support for tagging. STAR provides the required security functions without imposing significant performance and storage overhead using:

- Narrow tags for both instructions and data to support coarse-grained type-enforcement and to avoid illegal operations related to control flow, code and data pointers.
- *Directly-interpretable* instruction tags to enforce legal,

* *The first two authors share equal credit for the authorship of the paper.*

DOI 10.1109/SP46215.2023.00102

context-specific behavior of *existing* individual instructions, specifically for control flow and pointer integrity enforcement and to mark legitimate control flow targets in code. The direct interpretability of tags imply that instruction opcode decoding and tag decoding can be overlapped. Specifically, no table lookup using the (metadata) tag, as in [24], [26], [51], or special instructions with embedded instruction tags are required, as in [64]. The tagging of *existing* instructions in STAR also eliminates the need to add a number of usage-specific instructions to the ISA, reducing the design complexity.
- Novel instruction tag embedding within execution binaries to simplify instruction tag access.

Although the current paper focuses on control flow and pointer integrity applications, the basic tagging scheme of STAR can be adapted to encode various security-sensitive policies such as code checksum, statically determinable runtime invariants (e.g., last read/write of a variable) that target different threat models. These alternative uses are being investigated in our ongoing work.

STAR leverages the compiler as a rich source of information for providing the integrity of control flow as well as the integrity of data and code pointers in a general manner. Rich program semantics (e.g., pointer types) that are typically lost during compilation are captured by the STAR compiler and persisted into the binary in the form of instruction and data tags. These tags are processed and checked in the hardware to enforce the integrity of control flow, data pointers and code pointers to address vulnerabilities that stem from the inherent flaws in the software. This paper makes the following contributions:

- We present the design of a new hardware-based solution, which generalizes the use of narrow, directly-interpretable instruction tags and their use with data tags, in order to enforce control flow integrity and the integrity of code and data pointers. This is done by enforcing legal, context dependent behaviour of instructions.
- We introduce a novel instruction tagging scheme called inline tagging to eliminate the need for an additional cache for instruction tags, to co-access an instruction along with its associated instruction tag and thus maintain the locality of reference in accessing instructions.
- A full stack implementation of the proposed hardware, its compiler and OS support is presented, along with the security policies tied to the instruction and data tags.
- We demonstrate by running representative SPEC benchmarks that the average performance overhead against the baseline system is limited to low single digit percentages with a modest increase in the hardware resources.

The rest of this paper is organized as follows: In Section 2, we describe the threat model and assumptions. The instruction tags, data tags, security primitives and protection mechanisms of STAR are presented in Section 3. The hardware implementation details, compiler and OS changes are detailed in Section 4. In Section 5, evaluation of the scheme in terms of hardware design and software performance is presented. In Section 6, we present the related work and compare STAR against similar tagged hardware as well as other recent and relevant hardware mechanisms. Concluding remarks and future work are provided in Section 7.

2. Threat Model and Assumptions

We focus on preserving control flow and pointer integrity. We assume that the application may contain one or more software vulnerabilities that adversaries can exploit to launch a wide variety of attacks including control flow attacks. Specifically, these vulnerabilities can let an attacker gain control of the program stack to rewrite return addresses, code/function pointers, initiate buffer overflow attacks and bypass ASLR-protected memory using knowledge of the memory layout of the program. We assume the presence of Data Execution Prevention (DEP) in the hardware and that the code along with tags introduced by STAR can not be modified by the attacker.

This work addresses vulnerabilities that are limited to user-space, encompassing user applications and libraries. However, the solution can be extended to all software in a system, including the kernel.

The following additional assumptions are made:

- The hardware is free of inherent vulnerabilities. Therefore, we consider micro-architectural and side-channel attacks to be out of scope.
- The hardware platform enforces protection against code injection (e.g., NX-Bit). Many contemporary platforms deploy such a protection scheme by default.
- We assume that kernel, compiler, linker and loader are trusted components.
- We assume that the adversary has full knowledge of STAR's design details and the security protections do not depend on the secrecy of such information.
- While we do not explicitly rely on the presence of orthogonal defenses such as ASLR, it would only compliment and strengthen the protections offered by STAR.
- While our design is not fundamentally limited by ABI restrictions, in the current STAR system, legal pointers in memory are assumed to be 64-bit aligned.

3. Secure Tagged Architecture - STAR

We present the design of STAR using a proof-of-concept RISC-V implementation and describe the data tags, instruction tags, and security policies. Afterwards, we describe how these mechanisms work in conjunction to provide control flow and pointer integrity. Specifically, the goal of these mechanisms is to provide the following assurances, which constitute the foundational pillars of the STAR security framework:

1) Control transfers only take place between instructions specified in the software. These include function calls, jumps, branches and their respective targets.
2) Control flow targets are reached by the specific types of control transfer instructions designated for them. For example, function return instructions only transfer control to instructions identified as return targets.
3) All indirect control transfer instructions use valid code pointers.
4) Return addresses stored in memory are tamper-evident and cannot be used for control transfer if their integrity is violated.
5) Return addresses are created only by executing a function call instruction and cannot be forged by any other means.
6) Malicious overwrites to pointers (code & data) by non-pointer data, to code pointers by data pointers and vice versa are detected at the time of misuse.
7) Memory addresses used by load and store instructions are always valid data pointers.
8) Instructions not intended by the compiler to involve pointers cannot use, modify, or generate pointers.

Assurances 1 and 2 are provided by points-to analysis performed by the compiler in conjunction with instruction tags. Data type analysis along with instruction and data tags are used to provide Assurances 3 through 8. For the Assurance 6, the detection is delayed until the corrupted pointer is used to initiate an exploit. However, unlike silent errors where the source of the error is often unknown, because the precise violation is known during policy enforcement, the instruction that resulted in invalidation of the pointer can be identified, thereby aiding in diagnosis and triage efforts.

3.1. Data Tags

STAR tags each machine-word (32 bits aligned at 32-bit boundary) in memory with one of the 2-bit data tag values listed in Table 1. The data tags are symbolically represented using two capital letters surrounded by square brackets. Data tags enable code pointers, data pointers, and return addresses to be distinguished from one another and from other non-pointer data not only in memory but in registers as well. Data tags propagate from memory to register, register to memory, and register to register in accordance with the STAR security policies (see Section 3.3).

Tag	Description
[DT]	Non-Pointer Data
[DP]	Data Pointer
[CP]	Code Pointer
[RA]	Return Address

TABLE 1: Data Tags

3.2. Instruction Tags

STAR enforces legal, context dependent behavior of instructions through instruction tags. Each 32-bit instruction is associated with a 6-bit tag. An instruction's tag, in combination with the instruction's opcode, informs the hardware of the instruction's semantic context, and consequently the security policies to enforce.

The instruction tags used in STAR are presented in Table 2. The instruction tags are symbolically represented using three capital letters surrounded by square brackets. For showing the various possible instructions that can be associated with a particular tag value in a compact manner in Table 2, instructions are grouped into three categories. These categories are: control transfer instructions (function calls, function returns, indirect jumps, branches), memory instructions (load and store) and arithmetic instructions (addition, bit shifting, logical operations). In practice, the exact effect of an instruction tag depends on the instruction it is associated with. Thus, the same tag applied to a load instruction may produce a different behavior when applied to a store instruction.

Multiple instruction tags may simultaneously apply to a single instruction and are combined as a separate, single tag that combines the function of the constituent tags. For example, an arithmetic instruction that operates on a data pointer (tag [DPO]) may also be the target of a function call (tag [TFC]), as it is typically the case for the instruction that adjusts the stack pointer in function prologues. In this case, the single combined tag is represented as [TFC+DPO] and encoded to have the same width as each of the constituent tags.

Tag	Description	Applicable Instruction Group		
		Control	Memory	Arithmetic
[GEN]	Generic Instruction	✓	✓	✓
[DPO]	Data Pointer Operation		✓	✓
[CPO]	Code Pointer Operation		✓	✓
[RAP]	Return Address Push/Pop		✓	
[CLR]	Clear Instruction Source		✓	
[EQR]	Equal Rank Matching			✓
[CAL]	Function Call	✓		
[RET]	Function Return	✓		
[TFC]	Target of a Function Call	✓	✓	✓
[TFR]	Target of a Function Return	✓	✓	✓
[TIJ]	Target of an Indirect Jump	✓	✓	✓
[LBL]	CFI Label	Encoded as a no-op		

TABLE 2: Instruction Tags

3.3. Security Policies

We describe the security policies enforced by STAR for each instruction group separately.

3.3.1. Control Transfer Instructions. Every function call instruction is tagged [CAL]. The instruction that immediately follows the call is tagged [TFR]. The instruction at the target of the call is tagged [TFC]. Additionally, if the target function is unreachable within the compilation unit (i.e. its address is unknown at compile time), a CFI-label is inserted before the call in the form of a no-op instruction with the tag [LBL]. Likewise, each global function reachable by other compilation units has a matching CFI label at function entry points.

The CFI label is encoded as a no-op *lui* instruction that provides 20 bits of unused space, as shown in Figure 1. One bit is dedicated for the label type, which states whether the label is at the source of a control transfer or at the destination. Even though source labels are always followed by a call instruction, checking whether that is the case upon encountering a CFI label in the hardware would require a complex lookahead logic for reading the next instruction ahead of time. Since the hardware pipeline naturally fetches the instruction following the label in the next cycle, this check can be delayed until then without sacrificing security. Thus, the hardware uses the label type field to internally activate the check for the call instruction in the next cycle. The remaining 19 bits are used to implement a function-signature based CFI-scheme comprised of the function return type, first five argument types, and a flag to denote if the function is variadic. The CFI label itself is tagged [LBL], so that it can be distinguished from other no-op instructions.

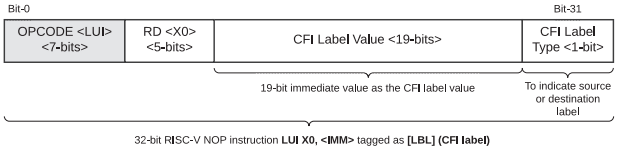


Figure 1: CFI label Format

When a call instruction executes, the register that stores the return address receives the data tag [RA]. From then on, only the memory instructions tagged [RAP] may load/store the return address. Furthermore, every return instruction is tagged [RET], which enforces that the register holding the return address is tagged [RA]. A return instruction can transfer control only to an instruction tagged [TFR].

Program counter relative (PC-Relative) jumps and branches have their targets verified at compile time and are not subject to additional security checks. On the other hand, every indirect jump instruction is required by the hardware at runtime to use a code pointer in its source register with the data tag [CP]. There is no instruction tag that identifies an indirect jump: any computed jump that is not a function call or a return, which are tagged [CAL] and [RET] respectively, is an indirect jump. For this reason, indirect jumps are tagged [GEN]. However, the instructions targeted by indirect jumps are tagged [TIJ] and the hardware enforces that the instruction following an indirect jump must be tagged [TIJ]. Note that direct jumps targeting an instruction tagged [TIJ] are still allowed; the target check is activated only upon encountering an indirect jump instruction.

In summary, the following protection mechanisms are provided to support control flow integrity:

- **Forward Edge Integrity:** Provided by pairing function calls tagged [CAL] and instructions at the target tagged [TFC] for functions resolved at compile time; by pairing compile-time unresolved functions using source and destination CFI labels tagged [LBL]; and by pairing indirect jumps tagged [GEN] with their potential targets tagged [TIJ]. Additionally, all control transfer instructions must use a source register tagged [CP] to ensure that the control transfer always happens using a code pointer.
- **Backward Edge Integrity:** Provided by pairing function return instructions tagged [RET] with their targets tagged [TFR]. All return instructions must use a source register tagged as [RA]. Morever, only the execution of an instruction with tag [CAL] will lead to tagging of the return address register [RA]. The data tag makes the return address tamper-evident. Only the memory instructions tagged [RAP] may operate on it. Any other instruction invalidates the [RA] tag and replaces it with [DT].

We formally define the security policies enforced by each instruction tag that applies to control transfer instructions using the pseudocode below. Each policy starts with the tag of the instruction, followed by the associated opcode and operands in RISC-V assembly notation. The *TAG()* operator returns the tag of a given entity, either a register or a memory location. The *MEM*| | operator denotes a memory location. When the *MEM*| | operator targets an instruction address, the expression *TAG(MEM*| |*)* denotes the tag for that instruction; otherwise it denotes the data tag. The ← operator denotes assignment.

- **Function Call Policy:**

```
[CAL] jalr ra, rs1, imm :
-------------------------------------
if TAG(rs1) != [CP]
  Security Violation
if TAG(MEM|rs1+imm|) != [TFC]
  Security Violation
TAG(ra) ← [RA]
```

- **Function Return Policy:**

```
[RET] jalr zero, ra, 0 :
-------------------------------------
if TAG(ra) != [RA]
  Security Violation
if TAG(MEM|ra|) != [TFR]
  Security Violation
TAG(ra) ← [RA]
```

- **Indirect Jump Policy:**

```
[GEN] jalr zero, rs1, imm :
-------------------------------------
if TAG(rs1) != [CP]
  Security Violation
if TAG(MEM|rs1+imm|) != [TIJ]
  Security Violation
```

Note that checking of the control transfer tags [TFC], [TFR], [TIJ], are done slightly differently in the hardware to improve performance. Instead of performing a memory operation to check the target instruction tag, a latch is set in the hardware that triggers the check while executing the target instruction in the pipeline. This latch is preserved across context switches as part of a process' context. Moreover, control transfer target checks are not triggered for fall-through code; only a control transfer instruction or a CFI label at the source activates the checks for the target. Additionally, a label based CFI check makes [TFC]

redundant for functions that have a CFI label at their entry points, since it is more fine-grained. In such cases, the STAR compiler doesn't tag the instruction at the entry point [TFC], and the hardware skips the check for the corresponding call instructions tagged [CAL].

3.3.2. Memory Instructions. The security policies for memory instructions focus on providing return address protection, pointer integrity, and information leakage mitigation between registers and memory. Generic load and store operations that don't require extra security checks are tagged [GEN]. The associated data tags propagate between registers and memory with the exception of the data tag [RA]: generic memory operations are not allowed to operate on return addresses. This is to say that a generic load cannot read from a memory location tagged [RA] and a generic store cannot store a register tagged [RA]. However, a generic store *can* overwrite a memory location tagged [RA] with a register tagged anything but [RA]. The [GEN] tag also enables helpers such as memcpy() to function properly without additional special handling.

When the STAR compiler determines that a memory operation loads or stores a data pointer or code pointer, it tags the instruction [DPO] or [CPO] respectively. The return address push and pop memory operations must be tagged [RAP]. When applied to a store instruction, it enforces that the source register must have the data tag [RA]. Similarly, a load instruction tagged [RAP] ensures that the tag of the memory location is [RA], before moving it into the register.

Additionally, memory instructions can be associated with the tag [CLR], which clears the source register and sets its data tag to [DT] for stores; and clears the contents of the memory location and sets its data tag to [DT] for loads. When used in combination with [RAP], it ensures that there is always only one copy of the return address at any given time either in a register or memory, preventing any potential abuse based on reusing the return address.

Support for setjmp/longjmp: Return address push/pop instructions without the additional [CLR] tag are used to support setjmp()/longjmp(), such that the same *setjmp* buffer can be reused by multiple longjmp() calls without invalidating the return address saved as part of the context.

Defense against use after free: The [CLR] tag can be used in a similar fashion on pointers before they are freed to limit the attack surface for use-after-free attacks. This is accomplished by tagging the pointer load instruction that precedes the free with [CLR], effectively invalidating the pointer in memory. We'd like to reiterate that no extra instructions need to be inserted to utilize this feature thanks to the STAR instruction tagging. However, coverage depends on the availability and the compiler's ability to identify applicable instructions where this feature can be leveraged.

In summary, the following protection mechanisms are provided by the memory-related security policies:

- **Return Address Integrity:** Provided by only allowing designated instructions tagged [RAP] to access the return address; by detecting any tampering of a return address through the data tag [RA]; by ensuring that the return address push/pop operations don't leave a copy behind in a register or memory for potential abuse through the use of [CLR].
- **Data Pointer Integrity:** Provided by ensuring that the memory instructions tagged [DPO] exclusively load/store data pointers tagged [DP]; by detecting tampering of data pointers through the data tag [DP].
- **Code Pointer Integrity:** Provided by ensuring that the memory instructions tagged [CPO] exclusively load/store code pointers tagged [CP]; by detecting tampering of code pointers through the data tag [CP].

Below is a summary of the security policies enforced by each tag that applies to memory operations:

- **Load Policies:**

```
[GEN] load rd, rs1, imm :
-------------------------------------------
if TAG(rs1) != [DP]
  Security Violation
if TAG(MEM|rs1+imm|) = [RA]
  Security Violation
TAG(rd) ← TAG(MEM|rs1+imm|)

[DPO] load rd, rs1, imm :
-------------------------------------------
if TAG(rs1) != [DP]
  Security Violation
if TAG(MEM|rs1+imm|) != [DP]
  Security Violation
TAG(rd) ← [DP]

[CPO] load rd, rs1, imm :
-------------------------------------------
if TAG(rs1) != [CP]
  Security Violation
if TAG(MEM|rs1+imm|) != [CP]
  Security Violation
TAG(rd) ← [CP]

[RAP] load ra, sp, imm :
-------------------------------------------
if TAG(sp) != [DP]
  Security Violation
if TAG(MEM|sp+imm|) != [RA]
  Security Violation
TAG(ra) ← [RA]

[CLR] load rd, rs1, imm :
-------------------------------------------
TAG(MEM|rs1+imm|) ← [DT]
    MEM|rs1+imm|  ← 0
```

- **Store Policies:**

```
[GEN] store rs1, rs2, imm :
-------------------------------------------
if TAG(rs1) != [DP]
  Security Violation
if TAG(rs2) = [RA]
  Security Violation
TAG(MEM|rs1+imm|) ← TAG(rs2)

[DPO] store rs1, rs2, imm :
-------------------------------------------
```

```
if TAG(rs1) != [DP]
  Security Violation
if TAG(rs2) != [DP]
  Security Violation
TAG(MEM|rs1+imm|) ← [DP]
```

[CPO] store rs1, rs2, imm :

```
if TAG(rs1) != [DP]
  Security Violation
if TAG(rs2) != [CP]
  Security Violation
TAG(MEM|rs1+imm|) ← [CP]
```

[RAP] store sp, ra, imm :

```
if TAG(sp) != [DP]
  Security Violation
if TAG(ra) != [RA]
  Security Violation
TAG(MEM|sp+imm|) ← [RA]
```

[CLR] store rs1, rs2, imm :

```
TAG(rs2) ← [DT]
    rs2  ← 0
```

3.3.3. Arithmetic Instructions. The STAR security policies for arithmetic instructions provide protections for pointer integrity and limited type safety. As data of a certain type in a register interacts with other registers, the type information must propagate appropriately. For example, an add instruction with the sole intention of adding two numeric values in separate registers must never operate on a pointer value, or output data of pointer type. A system that would prevent such vulnerabilities must also be flexible enough to allow an add instruction to either operate on two numeric values or a pointer and a numeric value (e.g. pointer increment). While doing so, it must accurately deduce at runtime that the former produces a numeric value while the latter a pointer value.

Data tag resolution in arithmetic instructions: STAR combines the instruction and data tags in a novel way to decide the tag of the output data and attempts to bridge the semantic gap between source code and binary. On the one hand, since the compiler has access to richer semantic information, it can identify the types of data that arithmetic instructions will operate on. On the other hand, the hardware has access to the type information in real time and can decide the target type effectively. Determining the data tag of an arithmetic instruction's output involves assigning a numeric rank to each instruction and data tag, calculating the rank of the output tag through a simple formula, and assigning the calculated rank to the output, as follows:

1) Determine the ranks for the instruction and data tags for each available source register using Table 3. Register *x0* in RISC-V is a read-only register, therefore its rank for any given instruction is the rank of the instruction tag itself from Table 3, ignoring the data tag column.
2) Calculate the RANK of the arithmetic output as follows:

$$RANK(output) = MIN(RANK(instruction), MAX(RANK(source\ register1), RANK(source\ register2)))$$

The RISC-V arithmetic instructions only allow up to 2 source registers. For arithmetic instructions with less than two source registers, omit the missing register(s) from the formula. When there are no source registers, omit $MAX()$ entirely.
3) Look up the tag that corresponds to the calculated rank from Table 3 and associate it with the destination register.

Instruction Tag	Data Tag	Rank
[GEN]	[DT]	0
[DPO]	[DP]	1
[CPO]	[CP]	2
—	[RA]	3

TABLE 3: Ranks for Instruction and Data Tags

The immediate observation from Table 3 is that there are no instruction tags with the rank 3, even though there is a corresponding data tag that is [RA]. The implication of this is that no arithmetic instruction may output a legitimate return address *even if* one of its sources is a return address. This makes intuitive sense as return addresses must remain immutable to maintain control flow integrity. On the other hand, if an arithmetic instruction tagged [CPO] adds an immediate value to the return address register tagged [RA] (i.e. [CPO] addi ra, ra, 4), the result *b*y design is going to be tagged [CP]. First, benign code does not perform arithmetic operations on return address. Second, even if such an instruction is encountered, the [CP] tag can not be abused unless the target of the code pointer is tagged as [TFC] or [TIJ]. Generalizing from this, one can observe that by being in charge of deciding the instruction tag, the compiler determines the maximum possible rank for the output while the hardware efficiently decides the precise data tag at runtime within the boundary set by the compiler.

This mechanism contributes to pointer integrity by ensuring that an arithmetic instruction not intended for pointer arithmetic cannot produce a pointer, while also ensuring that an arithmetic instruction intending to output a pointer may do so only if at least one of its inputs is a pointer. A stricter enforcement, where the instruction *must* have at least one input with the rank equal to the rank of the instruction, can be achieved with the tag [EQR]. This serves as an early detection mechanism for cases where an arithmetic instruction must operate strictly on a single data type and any deviation from that is illegal. Examples include: instructions that adjust the stack pointer (tagged [DP]) in function epilogue and prologue exclusively operate on data pointer type; instructions that increment/decrement integer loop variants exclusively operate on numeric values (tagged [DT]). Since the [EQR] tag can be toggled individually for each instruction, it provides the compiler with greater

flexibility in communicating the semantic information to the hardware.

In summary, the following protection mechanisms are provided by the arithmetic-related security policies:

- **Safe Pointer Arithmetic:** Provided by restricting the set of arithmetic instructions that can operate on pointers through the use of [DPO], [CPO] tags; by ensuring that instructions never intended to output pointers are not able to do so as per rank system and the [EQR] tag.

3.4. STAR Security Tagging in Use and Security Evaluations

Code reuse attacks like ROP, JOP and return-to-libc all rely on illegal pointer modification through the use of buffer overflows and other mechanisms. STAR's security tagging marks the pointers with the tag [DT] on such modifications and the original pointer tags are lost, so control flow instructions are unable to launch these attacks. Return addresses used to resume after a call can only be used if they remain tagged with [RA] in the register that holds them after retrieval from the stack. Gadget formation, relying on stringing together existing instructions, is further discouraged by allowing control flows into arbitrary instructions that are not marked as a target (using the tags [TFC], [TFR], [TIJ]) or to ones preceded by a label. Labeling indirect jump targets as described also ensures proper control flow into the intended targets from legal source instructions that do the control transfer.

The power of instruction tagging for defending against COOPLUS [14] attacks exemplifies how STAR is able to detect such attacks, even though the control flow does not violate the derived control flow graph. COOP [52] is a form of code reuse attack that does not require return address corruption in order to be initiated. Instead, COOP attacks hijack C++ objects and their *vptrs* in memory, and execute their existing virtual functions repeatedly in a carefully arranged manner. The hijacking is generally accomplished by exploiting a memory corruption vulnerability that enables the attacker to overwrite the memory area of an object. More advanced forms such as COOPLUS [14] can exploit polymorphism to invoke type-conformant, out-of-context, virtual functions wherever multiple transfer targets at virtual call sites are allowed. COOPLUS attacks are more sophisticated because the invoked virtual functions are part of the control flow graph as per C++ semantics, making detection more challenging for C++ semantic aware CFI solutions.

A typical COOPLUS attack works by corrupting a *vptr* of an object. The *vptr* of an object is a pointer to a table of code pointers (*vtable*) that are, possibly a subset of, the methods of a particular instance of a class. This dynamic indirect dispatch mechanism allows an inheriting class to override some of its base class' methods such that the *vptr* corresponding to those methods point to the derived class' implementation.

STAR can prevent such attacks thanks to its unique ability to combine instruction and data tags in a context-aware manner. As a motivating example, Figure 2 presents a C++ code sample, where a base class (called *Base*) declares a pure virtual method called *func()* and the two derived classes (*S1* and *S2*) provide separate implementations according to C++ semantics. Consequently, two separate *vtable*s for *S1* and *S2* exist, to which *vptr*s in the instances of these classes point to, respectively. The function *dispatch()* is an example of a virtual call site where transfers to both *S1::func()* and *S2::func()* are allowed due to polymorphism. The function *vulnerable()* contains a buffer overflow vulnerability in line 28 that overwrites the *vptr* of the object *s1*. The attacker's aim is to increment the value of the variable *data*. The COOPLUS attack overwrites this *vptr* with the *vtable* value of the class *S2*. Thus when *obj→func()* is invoked (line 19), *S2::func()* will be executed instead of *S1::func()*. The method *S2::func()* increments a member that does not exist in the class *S1*. Due to the layout of the stack in *vulnerable()*, the variable *data* gets treated as though it is *S2::memberM* and is incremented illegally.

The STAR compiler treats *vptr* as a data pointer and tags it [DP]. This is demonstrated in the assembly code for the constructor *S1::S1()* in Figure 2. Note that instructions are prefixed with their respective tags, and the inline instruction tags themselves are omitted in the output. The first three instructions calculate the *vtable* address in the hardware register *a0*. The very first instruction *auipc* is tagged [DPO], which tags the destination register [DT] based on the RANK resolution shown in Table 3. The last instruction *sd* in the code snippet stores the calculated *vptr* value in the object and is tagged [DPO], which ensures that the source register *a0* has the data tag [DT]. Overwriting *vptr* through buffer overflow changes its tag to [DT] (non-pointer data), similarly to overwrites to return addresses. When *vptr* is to be de-referenced to access the *vtable*, the corresponding load instruction requires the register containing the value of *vptr* (the address of *vtable*) to be tagged [DP]. This is demonstrated in the assembly code for the function *dispatch()* in Figure 2. The sequence of *ld* instructions perform the following: Dereference the object pointer to obtain the *vptr*; Dereference the *vptr* to obtain the base address of the *vtable*; Load the first entry in the *vtable*, which corresponds to the method *func()* of the corresponding object. STAR stops this attack at the first load instruction tagged [DPO], which requires that the loaded value from the memory has the data tag [DP]. Since the buffer overflow overwrites the data tag of the *vptr* with [DT], the check fails and a security violation is triggered.

STAR's security mechanisms were also tested against several microbenchmarks developed for evaluation, then with applicable RIPE benchmarks [2] and a subset of C-Bench [1] comprising of attacks involving indirect calls, return addresses and a COOP attack. In all cases, STAR's security mechanisms were able to successfully detect and prevent the attacks. The detected attacks were also identified by unique exception codes produced by the hardware and reported by the exception handler.

C++ Source Code

```
1  class Base {
2  public:
3    long memberN;
4    Base() : memberN(0) { };
5    virtual void func(void) = 0;
6  };
7  class S1 : public Base {
8  public:
9    void func(void) override { memberN++; };
10 };
11 class S2 : public Base {
12 public:
13   long memberM;
14   S2() : memberM(0) { };
15   void func(void) override { memberM++; };
16 };
17
18 // Call the derived object's func() method
19 void dispatch(Base *obj) { obj->func(); }
20
21 void vulnerable() {
22   long data = 42;
23   S1 s1;
24   char buf[8] = { 0 };
25
26   // Buffer overflow into the vptr of s1
27   // with the vtable value of the class S2
28   std::cin >> buf;
29   // Calls S2::func(), which increments the
30   // value of the stack-local variable data
31   dispatch(&s1);
32   ...
33 }
```

Tagged RISC-V Assembly - dispatch()

```
00000000000444f0 _Z6dispatchP4Base:
...
44514:  [DPO]    ld      a0, -24(s0)
44518:  [DPO]    ld      a1, 0(a0)
4451c:  [CPO]    ld      a1, 0(a1)
```

Tagged RISC-V Assembly - Constructor S1::S1()

```
00000000000446a0 _ZN2S1C2Ev:
...
446dc:  [DPO]    auipc   a0, 333
446e4:  [DPO]    addi    a0, a0, 1324
446e8:  [DPO]    addi    a0, a0, 16
446ec:  [GEN]    ld      a1, -32(s0)
446f4:  [DPO]    sd      a0, 0(a1)
```

Figure 2: A COOPLUS attack in C++, assembly snippets

4. Implementation of the STAR Architecture

4.1. Inline Instruction Tags

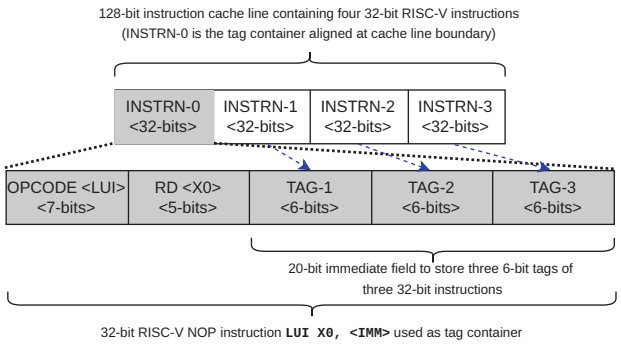
Instruction tagging requires extra storage to hold the tags. The current STAR implementation relies on the use of a 6-bit tag for regular 32-bit instructions in the RISC-V ISA. Instruction tags are co-located with instructions in the program executable, close to the instructions they correspond to. This is called *Inline Tagging* and eliminates extra memory accesses needed for locating and retrieving instruction tags had they been placed in a memory area separate and distinct from the memory area holding the executable. This particular choice of the tag width was made to enable instruction tagging to support security functions that go well beyond what is needed for enforcing the control flow and pointer integrity. Alternative choices exist for using wider instruction tags for more security measures or narrower instruction tags for restricted security applications - these are not discussed here for brevity.

The inline tagging scheme of the current implementation uses a single 4-byte tag field encoded as a no-op instruction within a cache line to cover three regular RISC-V instructions that follow the tag field. The tag field is aligned to start at a 16-byte boundary, making it simple for the tag to be located quickly within a cache line (Figure 3 (a)). The PC counter update logic is aware of the presence of the tag field within a cache line and automatically increments when it points to a tag field to point to the next instruction to be fetched. The compiler analysis phase generates the proper information to appropriately adjust offsets used for control transfers and instructions, in general. The performance penalty that comes from inlining instruction tags within the executable can be easily mitigated to acceptable levels (low single digit percentages) through the use of modest increase in the cache size and/or through the selection of cache parameters (Section 5.2). As shown in Figure 3(b), instruction tag and instruction opcode decoding proceed concurrently, without introducing any critical path that forces the use of a slower clock.

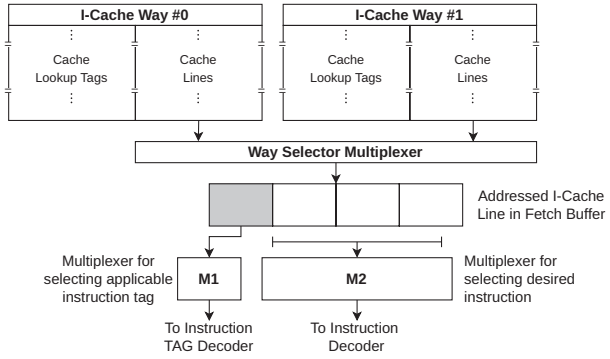
4.1.1. Generalized Instruction Tagging. The technique of inlining instruction tags within the binary is generalized and not restricted to a particular tag size or a particular I-cache line size. The general principle calls for placing the tags within a cache line in fields (called *tag fields*) of fixed sizes with pre-determined alignments. For instance, in a 32-byte cache line, the tags may be placed within a single 4 byte line, aligned at 32-byte boundaries. If the cache line is 64 bytes, two such fields can appear in a single cache line, or, in an alternative implementation, a single 8-byte field, aligned at 64-byte boundary can be used. Although the tag inlining is particularly suited for RISC ISAs that have uniform sized instructions, it can also be adopted for CISC ISAs.

The number of tags appearing in the tag field within an I-Cache line can also change based on the size of the tags assigned. With a fixed size for all instructions, the tags can have a fixed size and correspond to the instructions one-to-one that follow the tag field to the commencement of the next tag field. This is the approach we have taken in the STAR implementation described here, which is restricted to only the regular 32-bit instruction subset of the RISC-V ISA. To accommodate 16-bit compressed instructions, mixed in with 32-bit regular instructions, the tags for the former are kept at half the size of the regular 32-bit instructions to facilitate tag identification and placement within tag containers.

4.1.2. Narrower Instruction Tags. The tag assignment in the present STAR hardware prototype uses a flat namespace;



(a) Cache line showing a tag container



(b) Instruction tag access in a 2-way I-Cache

Figure 3: Inlined Instruction Tags

bit patterns in this namespace are allocated irrespective of the instructions they apply to. Functionally, the use of a particular tag is restricted to a few classes of instructions. Given this, one possibility is to interpret the tag depending on the opcode of the instruction. In the RISC-V implementations, opcode decoding proceeds in two steps – the first is the decoding to an instruction class (load, store, operation, control flow etc.), followed by the decoding of the opcode within the class. Opcode dependent tag interpretation introduces a short delay preceding tag decoding but uses tag namespace effectively.

4.2. Hardware Implementation Details

STAR is currently realized for a 64-bit RISC-V in-order pipeline on a Xilinx FPGA (Field Programmable Gate Array) development board (VCU-118). The baseline hardware, a Bluespec Flute pipeline [10], including caches, DRAM, I/O etc. are all realized on the VCU 118 board. The modifications needed to the baseline hardware enforcing CFI and pointer integrity are as follows.

The security checks for the tagging are realized using a set of separate stages that make up parallel pipeline ("Tag Processing Pipeline", TPP) with TPP stages adjacent to the corresponding baseline instruction pipeline's stages. The TPP logic for its stages include the necessary tag checks where needed. The hardware for the fetch stage is modified to skip the tag field in the I-Cache. The parallel pipeline approach somewhat eases the effort needed to port over the design to other RISC ISAs and out-of-order implementations.

A separate data tag cache (DT-Cache) is added to serve the L1-Cache for holding data tags. The L1 data cache (L1-DCache), the L1 instruction cache (L1-ICache) and the DT-Cache are all backed up by common level two cache (L2-Cache). A separate TLB (DT-TLB) is also added to accommodate the DT-Cache. The page table walking logic for handling TLB misses is replicated to handle DT-TLB misses. Memory accesses to serve a DT-Cache miss takes priority over handling a D-Cache miss, if any, that happens at the same clock cycle. The instruction cache (I-Cache) has been modified by adding an extra read port that reads the tag field in parallel to the instruction.

Each register in the baseline integer register file holds the data tag associated with its contents in a separate 2-bit tag register. All these data tag registers are held in a separate tag register file (TRF) within the TPP. A separate register file, the TSRF (*TPP State Register File*) holds the state of the TPP and any ongoing security checks, to support context switches. The TSRF also has registers that contain the addresses of the offending instruction that caused a security violation, along with a unique security violation code (SVC) for use by the security exception handler. The TRF and TSRF are also saved on any context switch and restored on return. Access to the TRF and TSRF are granted to special instructions that can only run in the S-mode (supervisor mode) of the RISC-V (where context saving and restoring takes place). These special instructions are implemented as metadata instructions, which are specially-tagged 32-bit inlined entities that appear as NOPS to the instruction pipeline. Other hardware changes used include the ability to update a data tag field in parallel with the normal data update part of the D-cache access. Together, these additions/modifications increase the hardware investments modestly (Section 5.2).

4.3. Operating System Support

The operating system is responsible for extending and enforcing the tagging support. The upper sixteenth of the userspace virtual memory area (VMA) is dedicated to hold the data tags that correspond to every machine word in the remaining lower address space. As a consequence, everything that used to be located in the upper sixteenth of the user VMA (stack, environment etc.) are slid down. The hardware locates the data tags for a given data item based on this modified memory layout. When setting up the initial execution environment for a process (for example as a result of an *exec* system call), the tags belonging to the data in the executable are mapped to their corresponding locations in the virtual address space. This step takes into account the address space layout randomization (ASLR).

Whenever a stack allocation causes a page fault that requires kernel to allocate more stack pages to the process, it also allocates the corresponding data tag pages if needed

and initializes them. Likewise, heap allocations that require more pages to be allocated to the process also trigger page allocations for the corresponding data tag pages when necessary and such newly-allocated tag pages are initialized by the kernel.

The hardware context of a process is extended to include the STAR-related registers. The saving and restoring of the STAR context is handled via accessing specific RISC-V CSR registers that map to the various STAR registers. These CSRs are only accessible in Supervisor Mode.

The virtual dynamic shared object (vDSO) is a mechanism that lets the kernel map specific system calls to the user space directly. The user program can invoke these routines within User Mode and avoid context switches, reducing the overhead of repeatedly calling frequently used system calls such *gettimeofday* and *rt_sigreturn*. Since the exported kernel code runs in the user space, it must be tagged appropriately. To generate the tagged vDSO library, we modified the Linux kernel build system to leverage the STAR compiler to build the vDSO. Currently, the STAR compiler does not support code written in RISC-V assembly. Thus, such assembly code was hand-tagged.

4.4. Handling Self-Modifying and JIT Code

In STAR, binaries cannot be overwritten, as embedded instruction tags can be overwritten to enable exploits. To support self-modifying and JIT code, the functions performing the modifications will have to be trusted and designed to insert the correct instruction tags. The self modifying code has to be tagged itself and access to it will have to be secured through control flow labels. To enable tagged code modifications in memory, a special handler will have to be supported, with the address of the self-modifying function pre-registered with it to enable the binary overwrites to proceed without a security exception. Software sandboxing can be used additionally to limit the range of memory locations affected by the self-modifying functions.

4.5. The STAR Compiler

We provide compiler support for STAR through a customized version of the LLVM 10.0.0 compiler due to its modular, reusable, and open architecture. The modifications build on a generic tagging compiler backend for RISC-V binaries [25] and can be broken down into three parts: the intermediate representation (IR) analysis within the LLVM common optimizer, the LLVM RISC-V backend, and the LLD linker. Figure 4 shows an overview of the toolchain.

Broadly, the static analysis for STAR is done at the IR level by a series of compiler passes. Data tags are assigned to global data, and instruction tags to individual IR instructions. These tags are then propagated through the lowering phase to the target specific backend. For both data and instruction tags, we rely on LLVM's usage, type, and address taken analyses. The latter is particularly important for tagging all possible basic block targets of indirect branches

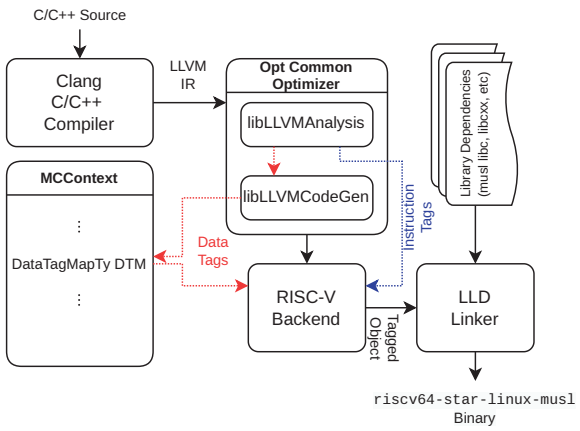


Figure 4: STAR compiler toolchain. Black arrows show the toolchain's data flow from source code to a tagged binary, blue and red arrows show data flow through the LLVM Pass Manager and the compilation unit's context.

with [TIJ] for supporting our CFI scheme. Note that target tagging is needed for destinations of indirect jumps; destinations of direct control flow transfers are not tagged. The inherent limitations of static analysis apply. Finally, the backend writes the instruction tags, cache-aligned, into the instruction stream during object file construction. For data tagging, global data objects are fully analyzed at compile time and their data tags are initialized by the data tagging pass. Data tags for local objects are set by instructions that create them consistent with their automatic, run-time allocation and de-allocation.

Global data tags are written into their own section in the object file, with a second section providing an index into that data tag section for each individual global symbol. Together these sections allow a modified LLD linker to resolve the proper tag to use with strong and weak symbols, and to write the unified tag section into the final binary.

5. Assessment: STAR FPGA Prototype and System

5.1. Prototype System Performance

The baseline processor used for the implementation described here is an in-order implementation of the 64-bit RISC-V ISA and follows the Flute design [10] very faithfully. This baseline processor has no special security features. The STAR processor is realized by adding a hardware subsystem to support and process instruction and data tags to the baseline design. A variety of STAR configurations were explored to examine different ways of mitigating the performance penalty that is imposed by the addition of the instruction and data tags and the need to fetch them into the processor for the security checks. Inlining instruction tags within the binary reduces the effective capacity of the I-Cache, which was used solely to hold instructions in the

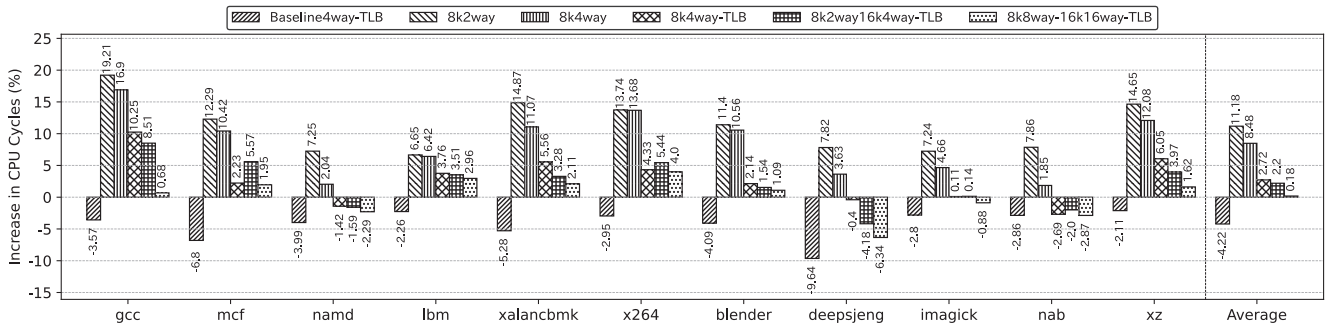


Figure 5: STAR performance overhead. The bars present the percentage increase in CPU cycles for SPEC-2017 rate benchmarks for all hardware configurations described in Table 4 over Baseline.

Configuration	Description
Baseline	No tagging, I-Cache and D-Cache are 8KB/2-way
Baseline4way-TLB	No tagging, I-Cache and D-Cache are 8KB/4-way, ITLB and DTLB have twice the capacity
8K2way	All caches (I-Cache, D-Cache and DT-Cache) are 8KB/2-way
8K4way	All caches (I-Cache, D-Cache and DT-Cache) are 8KB/4-way
8K4way-TLB	All caches (I-Cache, D-Cache and DT-Cache) are 8KB/4-way, ITLB and DTLB have twice the capacity
8K2way-16K4way-TLB	D-Cache and DT-Cache are 8KB/2-way, I-Cache is 16KB/4-way, ITLB and DTLB have twice the capacity
8K8way-16K16way-TLB	D-Cache and DT-Cache are 8KB/8-way, I-Cache is 16KB/16-way, ITLB and DTLB have twice the capacity

TABLE 4: Baseline and STAR Configurations

baseline processor. To reduce the performance impact from the higher I-Cache misses that result, the capacity of the I-Cache or its associativity (number of cache ways) can be increased. To fetch the data tags along with the associated data items for the security checks, a separate data tag cache (DT-Cache), operated in tandem with the D-Cache is used. Finally, to reduce the TLB pressure with inlined (instruction tags) and with the need to access the data tags, larger TLBs can be used for instruction, data and data tag accesses. The various configurations evaluated for the STAR processor, are shown in Table 4. The baseline processor supports a clock frequency of 100MHz. Likewise, all the synthesized hardware configurations run at the same clock frequency.

Figure 5 shows the performance penalty of the STAR configurations imposed by the use of tagging against the baseline design for the configurations studied. A baseline design with twice the TLB entries has also been included for comparison. The performance penalty is expressed as the percentage increase in the total number of CPU cycles spent in completing the benchmark execution, from start to finish. The data in Figure 5 were obtained from actual executions of representative SPEC-2017 benchmarks on the prototype FPGA implementation (on a Xilinx VCU-118 development board) running the STAR version of Flute at 100MHz, with 1GB of RAM, 8KB, 2-way L1 I-Cache, D-Cache and DT-Cache and a 1 MB shared L2 cache, using the modified LLVM compiler for tagging benchmarks and musl-libc on the modified Linux OS (5.4.10).

Based on monitored performance counters, I-Cache misses increased from 0.89% in the Baseline to 2.0 % in STAR's *8K2way* configuration, for compute-bound benchmarks like *xz* and *deepsjeng*. Simultaneously, L2-Cache misses increased from 0.98% to 3.91%, for applications with larger memory footprints like *mcf* and *lbm*. This increase in L2-Cache misses is due to the addition of the DT-Cache and inline tag placement. Increasing the cache associativity (*8K4way* configuration) improved the performance due to a reduction in the I-Cache misses, with a higher increase in the performance of memory-bound applications (like *mcf* and *x264*) compared to compute-bound applications (like *xalancbmk*, *deepsjeng*). Increasing the I-Cache size (*8K2Way-16K4Way* configuration), on the other hand, improved the performance of compute-bound applications as expected. Generally, memory-bound application like mcf, x264 and xz performed better with higher cache associativity while compute-bound applications like *gcc*, *xalancbmk* and *deepsjeng* performed better with I-Cache size increases. Doubling the TLB capacity (*8K4Way-TLB, 8K2way-16K4way-TLB* configurations) improved the performance of all the benchmarks because TLB walks were reduced. In the worst case, the hardware may have to do three page walks on TLB misses as the D-TLB performs the walks for both I-TLB and DT-TLB in the Flute implementation. These page walks increase the overall cache access latency. Another configuration (*8k2way-2MB L2*) was also examined by increasing the size of the L2-Cache, which improved the performance of all the benchmarks at the expense of increased area overhead of 95% and 20% for FPGA RAMB36 tiles and LUTRAM, respectively.

The results show that the overhead of instruction and data tagging can be best mitigated by doubling the TLB capacities instead of increasing the cache size and maintaining a modest 8KB data tag cache. This particular configuration (*8K4Way-TLB*) provides an average performance penalty of 2.72% across all the benchmarks studied, with the worst performance penalty at 10.25% for the gcc benchmark. To reduce the worst case and average performance overheads

Configuration	Dynamic Power Increase(%)	Total Power Increase(%)
Baseline4way-TLB	0.56	0.14
8K2way	4.8	1.34
8K4way	5.41	1.5
8K4way-TLB	5.88	1.64
8K2way-16K4way-TLB	12.55	3.57
8K8way-16K16way-TLB	14.6	4.13

TABLE 5: STAR: Relative power increase over Baseline

to 4% and 0.18%, respectively, the *8K8Way-16K16Way-TLB* configuration can be used.

5.2. Hardware Complexity and Power Assessment

The memory overhead for realizing STAR comes in the form of additional memory requirements for the data and instruction tags – this is inevitable. This overhead can be reduced by tailoring the tags to particular security applications and by using opcode-dependent instruction tagging (as described in Section 4.1.2). Within the STAR CPU, the added hardware needs come mainly in the form of the resources for implementing the DT-Cache, the additional resources needed for a larger I-Cache and/or TLBs (if used in the configuration) and the logic needed for the tag processing/security checks. The key indicators for hardware usage in the FPGA used for the STAR implementations are the memory blocks in the form of RAMB36 Tiles, and the LUT (look-up table logic) and the LUT RAMs - the latter two being primarily used for latches and logic associated with all components (including caches).

For the *8k4way-TLB* configuration, the usage of RAMB36 tiles go up to 298 from 269 for the Baseline system (about a 10.7% increase) and the LUT usage goes up to 123909 from the Baseline's 109,634 usage (13% increase), while the use of LUT RAM goes up from 2029 to 2265 (11.82% increase). For the *8k8Way-TLB* configuration, the usage of RAMB36 tiles go up to 357 from 259 for the Baseline system (about a 37.8378% increase) and the LUT usage goes up to 135,762 from the Baseline's 109,634 usage (23.832% increase), while the use of LUT RAM goes up from 2029 to 2269 (11.82% increase).

The reported increase in the hardware resources is for an in-order pipeline used in the Baseline and STAR. In a more likely scenario, where an out-of-order processor is used, hardware resource needs to support STAR's security tagging will be relatively much lower compared to the Baseline's for two reasons: (a) STAR's security mechanism is largely independent of whether the processor is pipelined or not, and (b) an out-of-order implementation of the baseline processor introduces several large hardware structures (like the reorder buffer, load/store queue with bypassing support, issue queue or reservation stations), larger branch predictors, separate physical and architectural register files etc. Thus, the additional hardware needed to add the security checks in STAR are fairly modest. As noted earlier, the storage overhead for STAR's tagging scheme can be reduced further to address specific security applications and through the use of opcode-dependent tag value assignment (Section 4.1.2). Certainly, if a higher performance overhead can be tolerated, the STAR hardware resource increases can be limited by using some of the other STAR configurations in Table 4.

The increase in power consumption of different STAR implementations over the (untagged) Baseline design are shown in Table 5. Both dynamic (switching power) and total power increases are obtained using the Xilinx power estimation tools. The total power increase is limited to slightly more than 4% in the most aggressive configuration.

6. Related Work

6.1. Control Flow Attacks and Software-Centric Solutions

Widely used hardware and software-based defenses against control flow attacks include Data Execution Prevention (DEP) [6], Address-space layout randomization (ASLR) [49]. However, these mechanisms are not adequate - for example, code reuse attacks can be launched even in the presence of DEP [61].

Matching the labels preceding a control transfer instruction with a counterpart at the legal target of the control flow has been one of the earliest software defenses against CFI attacks, as pioneered by Abadi [3], [4]. Traditionally, such label matching has been implemented in a variety of ways, including software-only mechanisms (compiler-based) [8], binary rewriting [67], virtual-machine based [68], and hardware assisted mechanisms. Usually, runtime checks are added to ensure that all function calls, jumps and returns are directed to valid locations that have been pre-determined prior to execution. The security and performance of these schemes depend on their implementations. Techniques that tag all the call sites with unique IDs as labels by analyzing all possible jumps are called fine-grained; these provide better security as they narrow down the possible control flow transfers. Unfortunately, such fine-grained techniques can have a performance overhead of 100% or more, as seen for stack protection [43] and other application scenarios [60], making them impractical. Coarse-grained solutions using control flow labels have lower performance overhead but offer weaker protection [40], [67], [68], and are vulnerable to attacks [12], [21], [29], [30].

In general, memory corruption can trigger control flow attacks. Protecting pointers on stack is not new and has been explored in many different ways. Low-overhead software-only stack protection schemes such as StackGuard [16] and shadow stacks [17] protect return addresses, but cannot protect other control flow structures. This makes them vulnerable to attacks that leak information and do direct writes to the memory. The overhead of software-implemented shadow stack mechanisms can be substantial [17]. Low-overhead memory safety techniques implemented in software have been proposed for code pointers [37]. However, broad-based, practical solutions for CFI and pointer integrity remain elusive.

System	Key Features	CFI Assurances					Comments
		RAP	**IJTE**	**OP**	**RPU**	**RAS**	
CHERI [63], [65]	Capability-based addressing to enforce principle of least privilege. 1-bit data tag distinguishes capabilities from other data. Memory safety is offered at coarse module granularity by protected calls across modules using capability to module.	◐ [a]	◐ [a]	◐ [a]	◐ [a]	◐ [a]	Significant software and system-level modifications required. Performance overhead of protected calls often very high [34].
PUMP [26], [27], DOVER [24], [59]	Wide metadata tags for both instruction and data. Metadata tag width can be as wide as the instruction or data they are associated with. Metadata tags are not directly interpreted; requires PUMP cache lookup prior to security micro-op activation.	●	●	●	●	●	Significant storange and performance overhead discourage deployment. Negative impact on clock cycle: 2-level PUMP cache lookup can exceed a single cycle in Level 1 (see text); 2-level PUMP cache complicates interrupt handling.
HDFI [56]	1-bit data-only tagging. Cannot distinguish between different types of pointers and non-pointers.	◐ [b]	○	◐ [b]	○	○	Only one specific pointer type can be protected at a time.
LowRISC [39], [57]	2-bit assignable data tag can implement CFI-related memory corruption protection.	●	○	●	○	○	Support is primarily for pointer protection and not all CFI attacks. New instructions needed to access pointers in memory.
ARM Pointer Authentication [41]	Sensitive pointers protected by cryptographic signature, requiring ISA extension.	◐ [c]	○	◐ [c]	●	○	Too expensive to protect all pointers.
ZERØ [69]	Memory protection for pointers with tagging that varies between L1 and L2 cache, requiring tag transformation. Load/store queues require changes for forwarding.	●	○	●	○	○	Support is primarily for pointer protection and not directed to all CFI attacks.
Intel x86 CET [54]	Return addresses protected using hardware-software shadow stack. Hardware-interpreted control flow labels track indirect control transfers.	●	●	○	○	○	Incomplete support for CFI. Shadow stack spills handled in software, introduces performance overhead.
Timber-V [64]	Lightweight enclaves using memory and metadata tags. New instructions for tag inspection (with expected tag encoded/embedded in loads) and manipulation.	◑	○	◑	◑	○	Significant software redesign needed to protect isolated pointers. CFI not tracked explicitly. Illegal pointer operations/use possible. Run-time overhead is significant.
STAR	Fine grained CFI and pointer protection through instruction and data tagging. Instruction tags are directly interpreted in parallel with the opcode to microoperations	●	●	●	●	● [d]	Tagging hardware extensible to add other hardware security features to avoid information leakage and implement protection domains (work in progress). Performance, storage and power overhead added by tagging is acceptably low.

TABLE 6: A comparison of tagged (and other recent) hardware protection features for CFI and pointer integrity. Abbreviations used in the *CFI Assurances* columns are **RAP**: Return Address Protection/Corruption Detection, **IJTE**: Indirect Jump Target Enforcement, **OP**: Other Pointers Protection/Corruption Detection, **RPU**: Restrict Pointer Usage to Legal Instructions, **RAS**: Reduced Attack Surface for Illegal Control Flow Transfers.
●: Supported, ○: Unsupported, ◐: Partially supported, ◑: not designed for CFI; too expensive to protect individual pointers and return addresses with enclaves
[a]: Protections offered at coarse granularity. For full protection, all pointers must be converted to capabilities and all calls must be protected.
[b]: Only one pointer type is protected, others are unprotected.
[c]: Too expensive to protect all return addresses and pointers; significant call/return overhead.
[d]: RAS through use of tags `TFC, TFR, TIJ`, when labels are not used.

6.2. Hardware-Centric Solutions

A plethora of hardware-assisted mechanisms have been proposed to overcome the performance overhead of software CFI and pointer integrity assurance techniques [15], [18]–[20], [22], [28], [31], [33], [35], [36], [38], [45], [46], [50], [56], [58], [66]. Hardware schemes include using shadow stacks to protect sensitive data [33], [48], tracing the control flow graph (CFG) of the program at runtime [5], [28], [31], [35], [36], [62], and using tagged architecture [11], [23], [24], [26], [51], [56], [64]. With addition of dedicated register files, the use of an additional label cache and hardware-supported label matching, the performance overhead of CFI can be reduced dramatically compared to software solutions. Hence, hardware support for shadow stacks have been proposed in [33], [48]. Although hardware stack protection is faster, providing memory safety for the full application is extremely expensive [42].

The use of tagged memory is an attractive choice for providing safety for pointers. The idea behind tagged memory is to extend each memory word with additional tag bits to protect against pointer (or data) corruption. Recent tag-based hardware systems and other recent mechanisms for CFI and

pointer integrity are summarized and compared in Table 6.

CHERI [63], [65] uses capability-based addressing in hardware to avoid illegal memory corruption and unintended information leakage. Capabilities are distinguished from other memory entities using a 1-bit tag. CHERI requires significant changes in the software layers and protected calls securing module boundaries have a significant performance overhead [34], making it impractical for fine-grained protection.

PUMP and variants [24], [26], [27], [59] use word sized metadata tags with individual instructions and data items, significantly increasing storage needs. The added need to look up a two-level PUMP cache using a wide key formed with the metadata tags and other information before any security micro-operations can be started introduces performance bottlenecks that can translate to a slower clock speed. Together, these drawbacks discourage any practical deployment. PUMP reports up to a 400% performance overhead using a simulator and is therefore not a real full stack implementation like STAR. We note that DOVER is a PUMP-derivative hardware implementation and is susceptible to similar performance bottlenecks as PUMP.

Data-only tagging as used in Hardware-Assisted Dataflow Isolation (HDFI) [56] has limitations in providing CFI, since non-control flow data can still be exploited to mount control flow attacks [32]. HDFI implements only one specific security mechanism at a time, while STAR supports a comprehensive set of CFI and pointer integrity checks. LowRISC [11] provides two bits of tagging in its memory system that could be used to implement a variety of memory safety mechanisms. ZERØ [69] is a pointer integrity mechanism that prevents corruption of code and data pointers in memory and relies on the use of new instructions for loading and storing these pointers. No protection is provided against illegal function or code pointer use by instructions, such as use of illegal indexing or use of illegal instructions that perform arithmetic and/or logical operations on these pointers and then use the modified pointers. In contrast, STAR uses instruction tags to ensure that pointers are accessed, modified and used only by instructions that are deemed legitimate by the compiler.

7. Conclusions and Ongoing Work

Enforcement of the context-specific behavior of existing instructions in an ISA can address many security vulnerabilities in software. The STAR architecture presented in this paper does so by using narrow instruction and data tags, with instruction tags embedded in the binary and colocated with corresponding instructions within a cache line, to realize a practically viable design. A set of instruction and data tags and their associated hardware-implemented security policies were proposed for STAR to mitigate control flow and pointer integrity attacks. A proof-of-principle hardware-software prototype of STAR was implemented on a Xilinx FPGA and appropriate supporting functions were incorporated within a modified Linux OS and a modified LLVM compiler. The performance, power and hardware overhead of the resulting system compared to an untagged baseline design are in the low single digit percentages; this improves the viability of deploying the system. STAR's tagging scheme goes well beyond the enforcement of control flow and pointer integrity and other tag-based security extensions to the architecture are being investigated as part of our ongoing work.

Instruction tagging, by itself, can implement other security functions without the use of data tagging and even the use of a single-bit data tag. Instruction tagging can also support applications beyond security, such as event-triggered code instrumentation, debugging and many others. We believe, that low-overhead instruction tagging, as introduced in this paper, will enable such applications.

8. Acknowledgements

This work is supported in part through a DARPA award under the SSITH program (Phases 1, 2 and 3) via DARPA Contract No. HR0011-18-C-0012, Office of Naval Research Awards #N00014-17-1-2929, NSF award #2047205, through a SUNY 2020 award and donations from Xilinx Corporation, with facilities support provided by the Center for Energy-Smart Electronic Systems at SUNY Binghamton. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] "CBench: Check Your CFI's Failed Protections," https://github.com/vul337/cfi-eval/tree/master/cbench.

[2] "RIPE 64-bit benchmark repository," https://github.com/hrosier/ripe64.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2005, p. 340–353.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[5] E. Aktas, F. Afram, and K. Ghose, "Continuous, low overhead, run-time validation of program executions," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 229–241.

[6] S. Andersen, "Changes to functionality in microsoft windows XP service pack 2," *Microsoft technical document, August*, 2004.

[7] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 227–242.

[8] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 353–362.

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011, pp. 30–40.

[10] "Flute open source code," https://github.com/bluespec/Flute, Bluespec Inc.

[11] A. Bradbury, G. Ferris, and R. Mullins, "Tagged memory and minion cores in the lowRISC SoC," *Memo, University of Cambridge*, 2014.

[12] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.

[13] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 559–572.

[14] K. Chen, C. Zhang, T. Yin, X. Chen, and L. Zhao, "{VScape}: Assessing and escaping virtual call protections," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1719–1736.

[15] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced control-flow integrity," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 2016, pp. 38–49.

[16] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX security symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[17] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, pp. 555–566.

[18] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-assisted flow integrity extension," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[19] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2014, pp. 1–6.

[20] L. Davi and A.-R. Sadeghi, "Building control-flow integrity defenses," in *Building Secure Defenses Against Code-Reuse Attacks*. Springer, 2015, pp. 27–54.

[21] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 401–416.

[22] A. De, A. Basu, S. Ghosh, and T. Jaeger, "FIXER: Flow integrity extensions for embedded RISC-V," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 348–353.

[23] A. A. De Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Micro-policies: Formally verified, tag-based security monitors," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 813–830.

[24] A. DeHon, E. Boling, R. Nikhil, D. Rad, J. Schwarz, N. Sharma, J. Stoy, G. Sullivan, and A. Sutherland, "DOVER: A metadata-extended RISC-V," in *RISC-V Workshop*, 2016.

[25] D. Demicco, M. Cole, G. Yuksek, R. T. Gollapudi, A. Prakash, K. Ghose, and Z. Umrigar, "Generic Tagging for RISC-V Binaries." arXiv preprint, 2022.

[26] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 487–502.

[27] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon, "Pump: A programmable unit for metadata processing," in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2611765.2611773

[28] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 131–148.

[29] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 901–913.

[30] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 575–589.

[31] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1470–1486.

[32] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[33] Intel Corporation, "Control-flow enforcement technology preview," https://www.intel.com/content/dam/develop/external/us/en/documents/catc17-introduction-intel-cet-844137.pdf, 2016.

[34] A. J. Joannou, "High-performance memory safety: optimizing the cheri capability machine," University of Cambridge, Computer Laboratory, Tech. Rep., 2019.

[35] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive call-site sensitive control flow integrity," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 95–110.

[36] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 195–211.

[37] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, 2018, pp. 81–116.

[38] J. Li, L. Chen, G. Shi, K. Chen, and D. Meng, "ABCFI: Fast and lightweight fine-grained hardware-assisted control-flow integrity," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3165–3176, 2020.

[39] lowRISC, "RISC-V LLVM," https://github.com/lowRISC/riscv-llvm.

[40] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity." in *NDSS*, vol. 26, 2015, pp. 27–30.

[41] A. Mujumdar, "Armv8.1-M Pointer Authentication and Branch Target Identification Extension," https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension, 2021.

[42] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 175–184.

[43] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *Proceedings of the 2010 International Symposium on Memory Management*, 2010, pp. 31–40.

[44] T. Newsham, "Format string attacks," https://people.cs.georgetown.edu/~clay/classes/spring2019/ia/format-string-attacks.pdf, 2000.

[45] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 577–587.

[46] B. Niu and G. Tan, "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1317–1328.

[47] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[48] H. Ozdoganoglu, T. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote, "SmashGuard: A hardware solution to prevent security attacks on the function return address," *IEEE Transactions on Computers*, vol. 55, no. 10, pp. 1271–1285, 2006.

[49] T. PaX, "PaX address space layout randomization (ASLR)," *http://pax. grsecurity. net/docs/aslr. txt*, 2003.

[50] P. Qiu, Y. Lyu, J. Zhang, D. Wang, and G. Qu, "Control flow integrity based on lightweight encryption architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 7, pp. 1358–1369, 2017.

[51] N. Roessler and A. DeHon, "Protecting the stack with metadata policies and tagged hardware," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 478–495.

[52] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 745–762.

[53] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.

[54] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '19, 2019.

[55] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 574–588.

[56] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 1–17.

[57] W. Song, A. Bradbury, and R. Mullins, "Towards general purpose tagged memory," in *Proceedings of the RISC-V Workshop*, 2015. [Online]. Available: https://riscv.org/wp-content/uploads/2015/06/riscv-tagged-mem-workshop-june2015.pdf

[58] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, "Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[59] G. T. Sullivan, A. DeHon, S. Milburn, E. Boling, M. Ciaffi, J. Rosenberg, and A. Sutherland, "The dover inherently secure processor," in *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, 2017, pp. 1–5.

[60] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[61] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 121–141.

[62] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2015, p. 927–940.

[63] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 20–37.

[64] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, "TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V." in *NDSS*, 2019.

[65] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 457–468.

[66] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 2012, pp. 1–12.

[67] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 559–573.

[68] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.

[69] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, and S. Sethumadhavan, "Zerø: Zero-overhead resilient operation under pointer integrity attacks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 999–1012.