TAG: Tagged Architecture Guide

SAMUEL JERO, NATHAN BUROW, BRYAN WARD, RICHARD SKOWYRA, and ROGER KHAZAN, MIT Lincoln Laboratory, USA
HOWARD SHROBE, MIT CSAIL, USA
HAMED OKHRAVI, MIT Lincoln Laboratory, USA

Software security defenses are routinely broken by the persistence of both security researchers and attackers. Hardware solutions based on tagging are emerging as a promising technique that provides strong security guarantees (*e.g.,* memory safety) while incurring minimal runtime overheads and maintaining compatibility with existing codebases. Such schemes extend every word in memory with a tag and enforce security policies across them. This paper provides a survey of existing work on tagged architectures and describe the types of attacks such architectures aim to prevent as well as the guarantees they provide. It highlights the main distinguishing factors among tagged architectures and presents the diversity of designs and implementations that have been proposed. The survey reveals several real-world challenges have been neglected relating to both security and practical deployment. The challenges relate to the provisioning and enforcement phases of tagged architectures, and various overheads they incur. This work identifies these challenges as open research problems and provides suggestions for improving their security and practicality.

ACM Reference Format:
Samuel Jero, Nathan Burow, Bryan Ward, Richard Skowyra, Roger Khazan, Howard Shrobe, and Hamed Okhravi. 2022. TAG: Tagged Architecture Guide. *ACM Comput. Surv.* 1, 1 (May 2022), 37 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

What some have called the "Eternal War in Memory" [107] between software defenses and attackers seeking to exploit software has raged for over two decades now. This war features numerous defenses developed by the academic software security community [1, 9, 18, 48, 74, 81, 83, 98], a subset of which are deployed by industry, namely W $\oplus$ X [83], stack canaries [24], randomization [58], and most recently Control-Flow Integrity (CFI) [1, 12, 124]. To date, each defense has been followed in turn by new attacks that bypass or sidestep the defense either by tailoring the attack technique to carefully avoid triggering the invariants checked by the defense, *e.g.,* return-oriented programming attacks [14, 39, 59], or by exploiting previously unknown vectors, *e.g.,* Row Hammer [112]. This progression is further exemplified by the memory-corruption domain, where deployed defenses such as W $\oplus$ X, CFI, or randomization [58], are in turn followed by new attack classes, namely code reuse [46], control jujutsu [39, 40], and information leaks [103, 115].

Authors' addresses: Samuel Jero, samuel.jero@ll.mit.edu; Nathan Burow, nathan.burow@ll.mit.edu; Bryan Ward, bryan.ward@ll.mit.edu; Richard Skowyra, richard.skowyra@ll.mit.edu; Roger Khazan, rkh@ll.mit.edu, MIT Lincoln Laboratory, USA; Howard Shrobe, hes@csail.mit.edu, MIT CSAIL, USA; Hamed Okhravi, hamed.okhravi@ll.mit.edu, MIT Lincoln Laboratory, USA.

0360-0300/2022/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

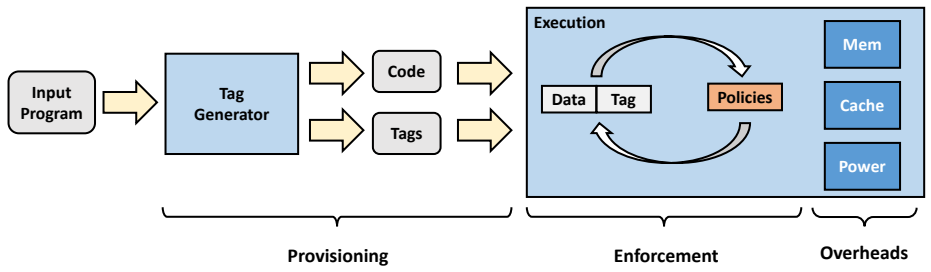

Fig. 1. The life-cycle of a tagged program

These attacks all arise because foundational security primitives have proven impractical in legacy software written in unsafe languages like C/C++, *e.g.,* runtime overheads on the order of 100% for memory safety [71, 92]. Consequently, there is renewed interest in security primitives built into hardware [72], with their promise of low or no overhead and strong safety guarantees.

Tagged architectures are a prominent class of hardware security primitives that augment data and code words with tags. Figure 1 illustrates how such architectures work. The tags, which function as the security metadata about memory, are created before the program is loaded. Then, at runtime, the hardware enforces security policies on the tags to provide safety guarantees, *i.e.,* checks the tags. Such checking comes with memory, cache, and power overheads. The benefit is that tags automate the secure and efficient management of security metadata, which has been the Achilles heel of software defenses [50, 56, 71], as illustrated by their performance overhead and attacks that target metadata [38].

Indeed, tagged architectures and associated policies have been developed to address the full gamut of software-security threats [100, 107], from type and memory corruption to integer overflows to thread safety. More concretely, tagged architectures for identifying and tracking pointers for memory safety [34], enforcing temporal memory safety [57], tracking type information [53], securing the stack and its return addresses [47, 90], enforcing control-flow integrity [37], taint tracking [37], tracking the flow of user input [86, 105], generalized information-flow tracking policies [17, 99], and a generalized pointer capability model [117] all exist.

At a high level, all tagged architectures and the policies they support share the same general goal: maintain metadata about the program's state at runtime, so that order can be brought to what some have termed the "sea of raw, teeming bits" [95] that currently makes up memory. Once a program has been compiled to a native executable, the vast majority of high-level semantic information about the program has been lost. Indeed, Von Neumman architectures [10, 114] deliberately conflate code, data, and pointers, making analysis even harder. Tags enable the processor to retain the security-critical subset of this lost semantic information, where the definition of "security-critical" varies from architecture to architecture, and may be user-specified.

To maintain the requisite security-critical semantic information in tags, tagged architectures augment the memory hierarchy (*e.g.,* main memory, caches, register files, *etc.*) with tag values, and modify the processor instructions to propagate tags. Policy is enforced either by inserting additional operations into the code using the native instructions of the machine, or by a

separate new engine added to the processor microarchitecture. While tagged architectures date back to the early 70's [76], they are undergoing a renaissance in the research community in recent years. To our knowledge, there have been 37 published efforts on tagged architectures over the past decade whereas there were only 20 efforts in the four decades preceding that.

In this work, we provide a survey of the tagged architectures themselves, including the threat model they address, along with the policy goal they achieve. However, for detailed discussion of software security policies, we refer to interested readers to existing work [100, 107]. We start by studying the historical influences and lineage of the modern tagged architectures. We then present a taxonomy for tagged architectures based on the important distinguishing factors we identify: *configurability*, *Trusted Computing Base (TCB)*, *tag implementation*, and *compatibility*. We further discuss the design choices and the trade-offs involved in implementing a tagged architecture. Building off the insights from our taxonomy, we identify challenges for tagged architectures in three broad categories: *provisioning*, *enforcement*, and *overhead*. We find that most of these challenges have received little attention from the community and little treatment in the related work.

A cross-cutting theme across the challenges we identify is that tagged architectures, despite their promise, remain a nascent technology. Security policies have yet to be verified. The processors supported are usually single core and in-order. To realize their potential as a deployable security technology, tags need to handle multi-core, out-of-order processors with features such as direct memory access (DMA). The interplay of hardware-based defenses, such as tags, with side-channel attacks has yet to be addressed. Evaluation of tagged architectures for performance, power, and area (PPA), while rapidly maturing, remains incomplete.

Our contributions are as follows:

- We study the historical influences and lineage of modern tagged architectures.
- We develop a taxonomy for tagged architectures and study their similarities and differences by reviewing the developments in this area over the past five decades.
- We discuss the design decisions and the trade-offs involved in implementing a tagged architecture.
- We pose and study the challenges faced by tagged architectures related to their security and practicality and find that there has been little work to address these challenges.
- We discuss some initial approaches for addressing these challenges, including existing hardware extensions, and illustrate that they are non-trivial open research questions.

2 TAGGED ARCHITECTURE OVERVIEW

Figure 1 illustrates the life-cycle of tags, from generation to their use at runtime. The tags themselves are simply metadata fields associated with, *e.g.,* words of memory or registers, which can encode arbitrary information. On top of the tags, the architecture encodes a set of policies that dictate how and when the tags are updated, propagated, and checked at runtime. For instance, a memory-safety policy might create tags on object allocation, propagate them to all pointers to the object, and check them on memory access, whereas an information-flow, *i.e.,* taint tracking, policy could create tags on IO, propagate them on copy, and check that control-flow decisions operate on untainted data.

Given the recent proliferation of hardware-security techniques, including support for bounds checking in Intel Memory Protection Extensions (MPX) [49], manipulating page-table permissions via Intel Memory Protection Keys (MPK) [79], and enclaves such as Intel SGX [22] and Sanctum [23], a more precise definition of tagged architectures is required. In this paper, we say that an architecture is tagged if it:

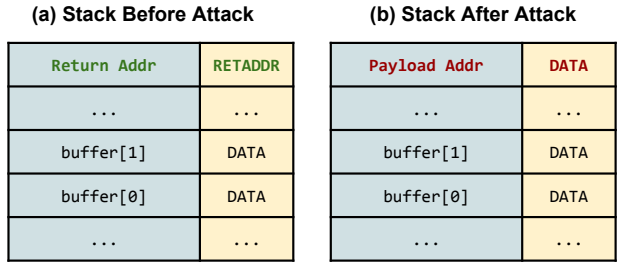

Fig. 2. Example of a tagged architecture enforcing a simple call-return policy.

(1) Supports metadata at the word granularity in memory,
(2) Ties tags to specific memory and/or registers, and
(3) Validates tags, producing errors for security violations.

Consequently, Intel's recent ISA extensions do not qualify as tagged: MPX does not tie tags to addresses; MPK is not word granular [89]; and enclave schemes (SGX) are a distinct security paradigm with an emphasis on isolating code / data, not on maintaining and validating metadata. Similarly, ARM's PAC [2] only maintains metadata for a subset of memory, pointers, and so does not qualify. However, ARM's MTE does meet our definition for a tagged architecture and is discussed here.

2.1 Threat Model

Tagged architectures and associated policies have been developed to address the full gamut of software-security threats [100, 107], from type and memory corruption to integer overflows to thread safety. However, not all tagged architectures address all threats, making it important to distinguish between the threat model addressed by a particular tagged architecture and the power of tags generally. For instance, some early proposals only targeted buffer overflows (spatial memory safety), whereas tags can address threats beyond memory safety. Indeed, tagged architectures that support user defined policies that can address arbitrary software security issues, including application logic bugs, are becomingly increasingly popular, taking the place of single purpose tagged architectures. The danger of such complex policies is that they significantly expand the attack surface introduced by tags to include policy errors. The guarantees provided by tagged architectures rely on the correctness of the policies they enforce, meaning that policy bugs are an important new threat introduced by tagged architectures.

Tagged architectures do not protect against attacks that target the hardware itself, including Row Hammer [112], Spectre [55], Meltdown [62], and Glitching [102]. Instead, tags ensure that the software layer above the hardware conforms to a given policy. Consequently, hardware side channels, speculative execution, and bit-flipping attacks are all out of scope. Attacks against device drivers or DMA, which are programmed by software, are in scope, but often left as future work.

2.2 Tag-Policy Example

A tag policy is a function that takes a set of tags as input and produces either a new set of tags or a violation, indicating that the tags break the expected invariant imposed by the policy. To see how tags and policies work in practice, consider guaranteeing the integrity of

return addresses to defeat classic stack-based control-flow hijacking attacks that overwrite them. A policy could be that only call instructions can write return addresses, and only return instructions can read them. To implement this, every return address is labeled with a `RETADDR` tag by the call instruction that writes it, with the rest of the stack receiving a default `DATA` tag. Instructions that write memory are updated to move the source's tag to the destination. Return instructions are updated by the policy to only execute if the return address still has the `RETADDR` tag.

The effect of this simple return-address policy is shown in Figure 2. If an attacker attempts to overflow a buffer to overwrite a return address, the input's `DATA` tag overwrites the original return address's `RETADDR` tag. When the program attempts to return to the attacker's payload, the modified return instruction reads the `DATA` tag of the return address and determines a violation. Note, however, that this policy does not stop *all* return-address attacks. The rule propagating tags on copy could be used by an attacker to move a return address further up the stack to the current function frame, allowing a stack unwinding attack. A stronger policy would be to disallow all writes to a return address *except* by call instructions. This issue highlights the importance of verifying that tag policies actually enforce the desired security properties. The usual suite of tools can be used for verifying tag policies, from formal methods to software testing techniques such as fuzzing.

3 TAG POLICIES

Existing tagged architectures have been used to explore five broad classes of policies: (i) *memory-safety* policies, (ii) *information-flow control* (*IFC*) policies, (iii) *dynamic information-flow tracking* (*DIFT*) policies, (iv) *capability models*, and (v) *programmable* architectures that support multiple policy paradigms, *i.e.,* are general-purpose. Memory safety and IFC specific policies were the first frontier of modern tagged architectures (since 2000), and specializations of them, *e.g.,* for embedded devices, are still areas of active interest [118]. The predominant area of interest, however, has been in DIFT policies that seek to limit the influence of user input in the program. Capability Models were revived by CHERI [117], and have seen significant interest in the community. Competing with the rise of capability models are general purpose tagged architectures, *e.g.,* PUMP [37] that seek to enable all possible tag policy paradigms. In addition to these main classes, the Typed Architectures [53] paper uses tags to support dynamic typing of data in languages such as JavaScript. This builds on the spirit of historical tagged architectures, such as the LISP machine [69] where the programming language and underlying architecture were tightly coupled.

3.1 Memory Safety

Memory-safety policies prevent memory-corruption attacks, *e.g.,* buffer overflows or use-after-frees, which lead to the attacker controlling the application, data leaks, privilege escalation, *etc.* Such architectures leverage the "fat-pointer" concept, associating bounds and version information with pointers and data, allowing the architecture to validate that a pointer is allowed to access a given piece of data at runtime. Exemplars include Hardbound [34], Watchdoglite [70], and Low-fat pointers [57]. Interestingly, commercial architectures such as the SPARC M7/M8 [75] and ARM MTE [3] generally focus on the bounds checking use case, though they provide limited tags and so can only provide probabilistic overflow detection. It is also possible to use the tag bits they provide for other policies, but given their focus on bounds checking applications, we currently consider them to support memory safety policies.

3.2 IFC

Classic IFC policies are concerned with the leaking of sensitive information, *e.g.,* crypto keys, or the use of privileged / classified information in non-privileged / unclassified settings. Consequently, they tend to use tags to enforce *e.g.,* Bell-LaPadula or Biba security models. More recently, this has been generalized to a notion of compartmentalization [66], with the goal of enabling least privilege for each compartment. Timber-V [118], the most recent IFC architecture, instantiates this as secure enclaves for embedded devices.

3.3 DIFT

DIFT policies are based on policing how user input, or data derived from user input, can be used to impact application data and control-flow. As such, they are a subset of IFC, but have seen sufficient research interest that we discuss them separately. DIFT policies address threats ranging from control-flow hijacking to data leaks, and even higher level policies such as compartmentalization within an application. Indeed, there exists a substantial body of work on formalizing security properties in terms of information flow [41]. Even memory safety has been formalized as a subset of IFC, by formulating memory safety as a non-interference property [31]. The effort to formalize security properties as IFC has run into issues in practice however. In particular, using pointer tainting (a subset of DIFT) is known to be ineffective at detecting data leaks and memory corruptions of non-control data [96], though there is some debate about how severe these limitations are [27]. In part because of these debates, and in part because of difficulties in scaling DIFT policies without large number of false positives or false negatives, current research emphasis has moved away from DIFT towards capabilities and general purpose tags.

3.4 Capabilities

The CHERI capability model has been well explored within the literature by the CHERI team [21, 28, 116, 117, 121]. The capability paradigm has also seen broader adoption within the security community, including notably seL4 [54]. CHERI offers two types of capabilities: object capabilities that allow the programmer to encapsulate code and data, and memory capabilities added by the compiler. CHERI is also the most mature tagged architecture with academic roots, and ARM intends to fabricate CHERI boards, to be available in 2021 [19].

3.5 Programmable policies

General-purpose tags [4, 37, 101, 106] may be more flexible than the other paradigms, as they can be interpreted according to any model, including capabilities. Common policies include Control-Flow Integrity (CFI), Code-Pointer Integrity (CPI), Shadow Stacks, Bounds Checking, Use-After-Free checks, and IFC like policies for, *e.g.,* taint tracking. However, general-purpose tags require significantly more programmer support to write specialized policies, as well as sharing a reliance on compiler support for the initial tags. Regardless of programming paradigm, general-purpose tag policies should be verified to show that they enforce the desired security properties. Even if the policies themselves are amenable to analysis [29], users are still dependent upon a large and complex TCB to determine the initial tag set given as input to the policy at runtime.

4 HISTORICAL EVOLUTION AND INFLUENCES

Perhaps the earliest incarnation of a tagged architecture was the B5000 series machines built by the Burroughs Large Systems Group in 1961 [76]. It was the first departure from the

conventional von Neumann architecture in that it used a 3-bit tag to differentiate between code and different data types. It was the first significant effort to enrich the processor with semantic information, albeit in a limited fashion. A fascinating aspect of this design is that even though it provided a limited form of memory safety, security was in its infancy and many forms of attacks that the B5000 series machines prevented by virtue of its semantic information would not be understood for another few decades. For example, a stack-based buffer overflow is prevented in the B5000 machines because they distinguish between return addresses and user data, but such attacks would only be studied in any meaningful depth in the 90s [73]. The tags in the B5000 machines were primarily a way of providing 'information structure' for a computer system and were utilized to provide some level of hardware fault tolerance. The B5000 machines are also the first to use the term 'tag' for this extra semantic information.

The paper by Feustel [42] in 1973 more systematically identifies the benefits of tagged architecture, but again, they are again viewed as a way of providing more semantic information in hardware in order to enrich and simplify programming language and operating system designs and make them extensible. Feustel hints at the possible security advantage of tagged architectures (called 'protection'), but they are viewed as a way of enforcing access for shared resources.

The LISP machine [69] circa 1985 is perhaps the first to study the security advantages of tagged architecture in a way that resembles modern understanding of them, so this is the point that different policy goals diverge from each other. Accordingly, we break down the subsequent historical development of the tagged architecture into their own subsections for the rest of this section.

Figure 3 illustrates the historical influences and the lineage of tagged architectures. Each color denotes a policy goal: blue is memory safety, yellow is IFC, green is DIFT, and orange is programmable. An arrow from A to B denotes A directly influencing B. A dotted line is an indirect influence. Stripped coloring shows an improvement or specialization of a design. Ribbon shape denotes a seminal or influential design.

4.1 Memory Safety

As mentioned, the LISP machine and its incarnation in the Symbolics 3600 [69], are among the first to utilize tags in a way that resembles modern understanding of them. Symbolics 3600 used tags to facilitate a relocating garbage collector and enable "run-time checking of data types" [69]. At the very least, this machine provided temporal memory safety and could prevent some forms of variable and stack overflow depending on the actual checks implemented on top of tags. A 'stack overflow' is when the stack contains too many frames and bleeds into adjacent memory, and it should not be confused with 'stack-based buffer overflow' that will be studied and exploited for control-flow hijacking later in the 90s.

Influenced by Symbolics 3600, the SPUR Lisp [125] is the next machine that used tags to facilitate garbage collection. Unlike Symbolics 3600, the SPUR Lisp machine stored all types of data in the same area of memory to achieve better alignment, thus it is among the first to propose a flat address space that leverages tags to distinguish between various data types. SPUR later influenced KCM [6] that, although it was a Prolog machine, used tags for garbage collection in a very similar fashion. The usage of tagged architectures for memory safety went into a winter for two decades.

The seminal work that heavily influenced modern tagged architectures for memory safety was HardBound circa 2008 [34]. HardBound is notable form multiple perspectives. By 2008, spatial and temporal memory safety violations were well-understood in the community [16,

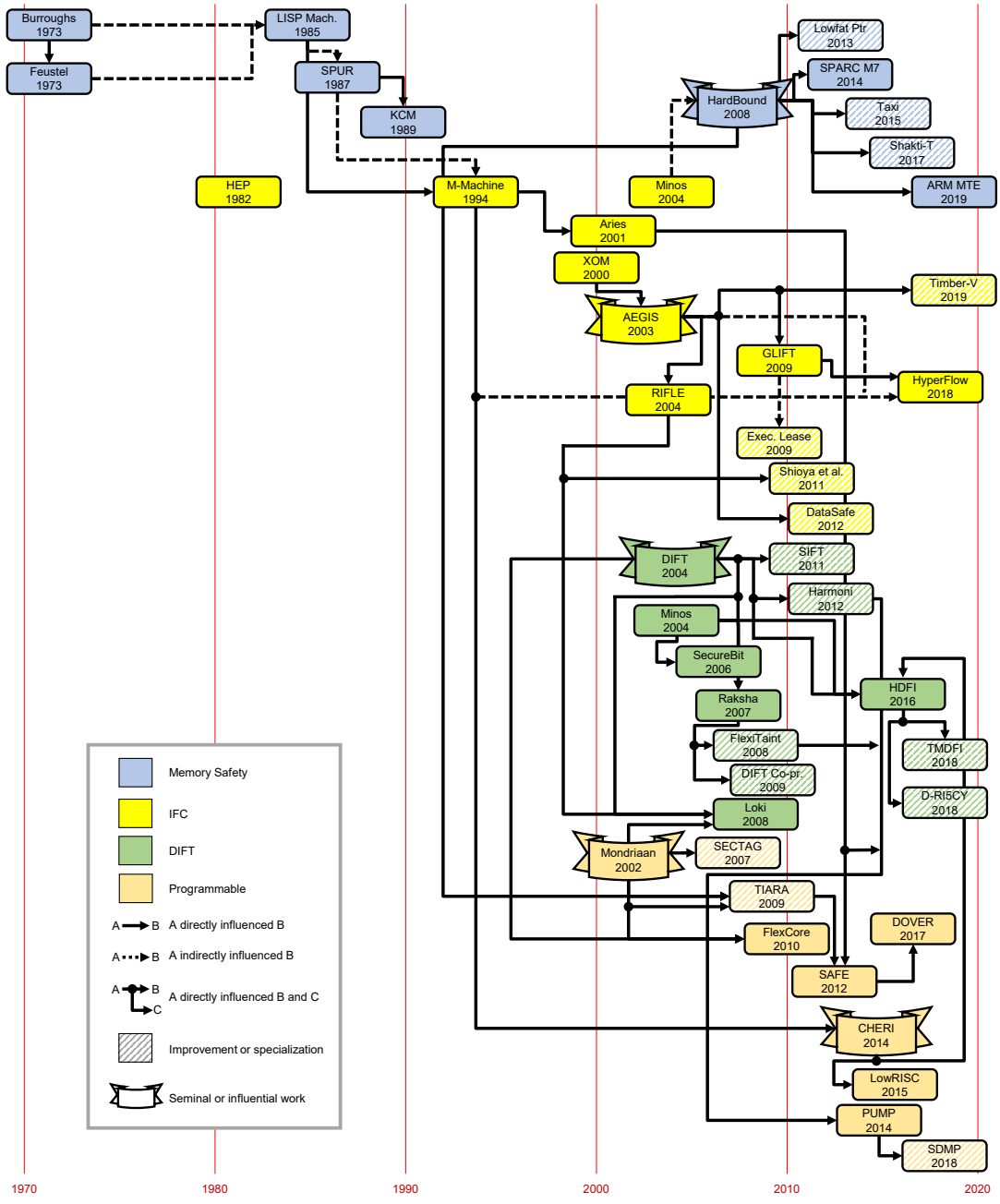

Fig. 3. Historical influences and lineage of tagged architectures categorized by their policy goals.

73, 122] and their potential for control-hijacking attacks was observed in the wild [44]. As such, HardBound is perhaps the first tagged architecture that is motivated by preventing what is today understood as the broad class of memory safety bugs, and was not from the lineage of machines that merely facilitate garbage collection. HardBound was also the first to explicitly encode base and bounds for spatial memory safety in tag bits and provided a concrete realization of such run-time checks.

HardBound heavily influenced low-fat pointers [57] that encode a compact version of base and bounds metadata in tags for better performance and memory overhead, and SPARC M7/M8 [75, 84, 88] that use smaller tags (also called 'color') for probabilistic overflow detection but provide lower silicon overhead. In addition, Taxi [43, 47] specifically studies policies that can protect against code reuse attacks. Shakti-T [67] further optimizes storage requirements for tags by using a common memory region for all base and bounds information. Finally, ARM MTE [3] presents one of the latest commercial incarnations of memory-safety-focused tagged architectures influenced by HardBound, but MTE also takes some influences from SPARC in its notion of coloring because of its small tag space (4 bits).

4.2 IFC

An early form of IFC was implemented by HEP [97] in1982 to prevent interference of processes running in parallel on a multiprocessor computer system. Modern notions of information-flow control and non-interference did not exist at the time, so the discussions in HEP are all centered on *enabling* parallelism. M-Machine [15] is perhaps the earliest incarnation of a tagged architecture that provides non-interference for security reasons. Influenced by the LISP Machine's idea of using tags to identify pointers and advocating the idea of a flat memory space (similar to SPUR), the M-Machine uses tags to encode what region of memory a pointer can access. M-Machine calls such pointers *capabilities*. M-Machine thus provides non-interference among multiple processes that share the same address space, while avowing the cost of context switches for permission checks. Note that non-interference is the simplest form of IFC where no exchange of information among various compartments (in this case, processes) are allowed.

M-Machine influenced Aries [8] that extends the flexibility of its policy by enforcing a lattice-based IFC such as the Bell-LaPadula model (*e.g.,* no reading from a higher classification and no writing to a lower classification).

An independent work in this area, XOM [61], leveraged encryption to protect data (and code) in memory from interference from other processes. Tags are used to identify the compartment of data at runtime. XOM allows sharing of data among processes by explicit key sharing, thus it enforces a richer IFC policy that allows sharing.

Another influential design in this domain was AEGIS [104]. Itself influenced by XOM, AEGIS was the first to develop the idea of protecting a part of an application in its secure mode and removing interferences with that part. In modern terminology, this is called an *enclave*. AEGIS used tagging to prevent interference in on-chip caches, but it did not allow shared memory. AEGIS influenced work such as RIFLE [111] that enforced a lattice-based IFC, GLIFT [110] that extends tags to all logical gates in the machine (instead of just data path), and execution leases [109] that improve the flexibility and performance of GLIFT by granting control of a portion of the machine to untrusted code for a fixed amount of time. Shioya et al. optimize the memory and latency overhead of a design like RIFLE [93]. DataSafe [17] also implements a protected enclave, but provides a more expressive policy and automatically transforms binaries without requiring source code; it is thus an improvement over a design like RIFLE.

One of the latest and most mature designs in the IFC lineage is HyperFlow [41]. HyperFlow has direct inspirations from execution leases and GLIFT and perhaps indirect influences from many other previous IFC work such as M-Machine and AEGIS. HyperFlow is an expressive IFC implemented on top of a complete RISC-V processor and is capable of enforcing arbitrary lattice-based policies. TIMBER-V [118] implements AEGIS-like enclaves for embedded systems on top of RISC-V and provides efficient inter-domain communication. TIMBER-V leverages tags to enforce domain isolation rather than enforcing a generic IFC policy the way HyperFlow does.

4.3 DIFT

Tagged architectures that enforce DIFT have not only seen substantial research particularly in the period between 2004 to 2018, but also they have had many influences on other designs for policy goals, speficially memory safety and IFC.

The seminal work in this area was the DIFT [105] paper by Suh et al. that not only defines the concept of DIFT but also illustrates how various security policies can be realized as a DIFT policy and implements a tagged architecture for enforcing it. DIFT might have had influences from early tagged architecture designs, but its ideas sparked a new line of research for tagged architectures, sufficiently different from IFC and other security goals that we consider it the inception of that domain.

Minos [25] developed in parallel with DIFT enforces Biba's low water-mark integrity policy on a tagged version of Pentium emulator. Although it is a form of dynamic information-flow tracking, Minos's policy is fixed and not as expressive as the work by Suh et al.. Minos influenced SecureBit [86] that leverages a DIFT-like policy to prevent buffer overflows to control data.

DIFT work by Suh et al. and Minos influenced Raksha [26] that implemented a tagged SPARC FPGA to enforce flexible and programmable DIFT policies. FlexiTaint [113] generalizes the propagation rules in a design like Raksha. DIFT co-processor [52] simplifies hardware design and verification for a design like Raksha by enforcing the policy in a co-processor. Loki [123], influenced by Mondriaan [119, 120], DIFT [105], and RIFLE [26] enforces application separation without the need to trust the OS. It also supports a large number of protection domains. SIFT [78] implements a low-overhead version of DIFT by leveraging a separate thread to perform taint propagation and policy checking. Harmoni [33] optimizes the efficiency of enforcing DIFT by implementing a hardware accelerator.

A major advancement in this area was achieved by HDFI [99]. Influenced by Suh et al. [105] and Minos [25], HDFI implements an full-blown RISC-V processor on FPGA capable of enforcing a rich set of DIFT policies. It also leverages compiler augmentation to emit DIFT instructions. TMDFI [63] improves upon HDFI by having larger tags. Finally, D-RI5CY [80] presents a low-overhead version of DIFT for IoT applications.

4.4 Programmable

Programmable tagged architectures provide a multi-purpose tag logic that can be used to enforce the other specific policies described above: memory safety, IFC, and DIFT. Although rare in early years, programmable tagged architectures have seen a major surge in recent years, particularly after 2012.

The seminal work in this domain was Mondriaan [119, 120] that influenced many modern designs. Mondriaan implements a fine-grained memory protection mechanism that tags individual words in memory with permission bits. Efficient lookup and enforcement of permissions along with cross-domain accesses secured by 'gates' allow Mondriaan to enforce

a rich and programmable set of policies. SECTAG [4] implements a similar but more limited design that was only validated in simulation. Influenced by Mondriaan and HardBound, TIARA [94] specifically enforces programmable policies to minimize the privilege in the operating system. FlexCore [32] was influenced by Mondriaan and DIFT and used a design that combines an FPGA fabric with ASIC to enforce a programmable policy efficiently.

A distinct work in this domain was the SAFE processor [5, 36] that was influenced by TIARA and Aries. SAFE not only implemented an efficient and programmable tagged architecture, but also it was notable in being one of the first efforts to formally verify its tag policy. The SAFE processor later evolved into a significantly more feature-rich and flexible tagged architecture called DOVER [106]. DOVER is notable in 1) being among the first to have a domain-specific language for policies that automatically translate to tag rules and 2) utilizing a separate core (called PEX) for policy enforcement. This is unlike many past and follow-on efforts that use 'in-pipeline' modifications to enforce the policy. DOVER is capable of enforcing a rich set of programmable policies.

A major advancement in the area of programmable tagged architecture and one that influenced almost every subsequent work was CHERI [116, 117, 121]. CHERI itself was influenced by the M-Machine and its notion of capabilities. CHERI implements a tagged architecture to keep track of capabilities, a set of compiler transformations to implement pointers as capabilities, and a rich software stack that leverages such capabilities to implement a wide range of security policies from memory safety to high-level application policies. CHERI and its compoenents are described in a series of papers [19, 21, 28, 116, 117, 121] and represents an influential portion of the literature on tagged architecture. CHERI is under active development and transition to practice at the time of writing this survey.

CHERI inspired LowRISC [64, 101], which is a practical implementation of programmable tags on the Rocket RISC-V chip, but its implementation seems to be incomplete at the this time.

Another work that took inspiration from Aries, FlexiTaint, and Harmoni was PUMP [30, 35, 37]. PUMP also provides an expressive and general-purpose tagged processor capable of enforcing programmable policies ranging from memory safety to IFC and DIFT. PUMP is notable in that, similar to SAFE, it performs formal verification on tag policy [30]. We will discuss the implications of complex tag policies for their correctness in Section 8. Verifications such as those performed by PUMP provide more assurance about tag correctness, and with the growing complexity of tag policies, they seem to be an essential aspect of tagged architectures in the future. Lastly, SDMP [90] specializes PUMP for stack protection using various policies.

5 TAXONOMY

For each architecture, we present its policy goal along with a classification of its implementation in a four-dimensional taxonomy for tagged architectures: Configurability, Tag Implementation, Trusted Computing Base (TCB), and Compatibility. These four dimensions highlight the important differences among tagged architectures in terms of the policies that they support and the cost of deploying them. In particular, the architecture's implementation and its Configurability go hand in hand and directly affect what policies can be supported. As the motivation for tagged architectures is to provide hardware acceleration for security policies, policy support is a critical factor when comparing them. Detailing the TCB provides a nuanced view of the security guarantees possible from a tagged architecture, while Compatibility looks at other barriers to adoption. We present a summary of all architectures

we examine in Table 1. Here we provide a brief of overview each dimension, with further details in the following four sections.

Configurability. Tagged architectures can support a fixed policy, a limited set of policies, or be fully programmable. Single policy architectures are simpler and easier to implement, whereas fully programmable architectures offer flexible protection against a wider ranger of threats.

Tag Implementation. The key components of a tagged architecture's implementation are how big the tags are, how the tags are managed, and how they are stored. In turn, these determine the type of policies that can be supported, and at what runtime and memory overhead.

Trusted Computing Base. The TCB of a tagged architecture necessarily includes the processor, and depending upon the architecture may also extend to the compiler, bootloader, operating system, and main memory.

Compatibility. Tagged architectures fall into several categories with respect to compatibility with existing code bases: no compatibility, binary compatibility, and source compatibility. This needs to consider not only whether binaries will run but whether they will receive security benefits.

5.1 Configurability

In support of their policy goal, tag architectures can either be completely fixed and non-configurable, or expose a variety of interfaces to policy designers. Non-configurable architectures support a fixed policy that is "baked in" to the silicon [6, 8, 25, 47, 53, 61, 67, 69, 76, 86, 93, 97, 104, 109, 110, 118, 121, 125]. These designs were common in early tagged architectures, and still manifest as a lightweight means for enforcing policies. Of the configurable architectures, we identify two different classes of configurability, directly analogous to whether the architecture support a programmable policy or not. Where the architecture does not support a programmable policy, the tags are less configurable [26, 41, 52, 63, 78, 80, 111]. Recently, the trend is towards fully programmable tagged architectures [17, 32, 33, 36, 37, 90, 95, 101, 106, 113].

Non-Configurable. Fixed policies are attractive because they are invariant across all deployments of the system, so one can reasonably make security claims for all code running on the processor. The disadvantage is that they are inflexible – their security model cannot evolve to address new threats. In the 70's and 80's, the original tagged architectures were invented to provide native support for languages such as LISP or Prolog [6, 69, 125], or unique general-purpose machines that aimed to disambiguate code/data/pointers [76, 97]. The next wave of tagged architectures, starting in the 2000's, were explicitly security focused, with interest in IFC [8, 47, 86, 93, 104, 109, 110]. Along with interest in taint tracking via IFC came interest in data integrity/confidentiality policies such as Biba and Bell La Padula [25], and enclaves that were resistant to code leakage (and hence software piracy) [61]. The most recent development in fixed policies is a return to language support, this time for dynamically typed languages by introducing new ISA primitives that are aware of the type of the data at runtime and execute correctly [53].

Note that most tagged architectures that focus on memory safety [15, 34, 57, 67, 84, 120] are not programmable, and typically rely on the tag to distinguish pointers and data, Hardbound [34] is a prominent example of this. Further, such architectures are comparatively rare and may be incompatible with common C paradigms such as out-of-bounds pointers.

Table 1. Tagged architectures with their design dimensions sorted by the publication year. **Policy Goal:** * – spatial memory safety only, † – Optimization work rather than a new architecture. **Tag Impl:** W – widened memory, D – disjoint memory, PT – page-table-like structure, LT – lookup table structure, E – memory encrypted with unique key for each tag. **TCB:** C – compiler, B – bootloader, P – processor, K – OS kernel, M – main memory.

Architecture	Year	Policy Goal	Config?	Tag Impl			TCB					Compat	Evaluation
				Size (bits)	Mgmt	Store	C	B	P	K	M		
Timber [118]	2019	IFC	No	2	ISA	W	●	●	●	●	●	Source	Simulation
ARM MTE [3]	2019	Memory Safety	Yes	4	ISA	N/A	●	●	●	●	●	Binary	ASIC
D-RI5CY [80]	2018	DIFT	Yes	1	API	W	●	●	●	●	●	Binary	FPGA
TMDFI [63]	2018	DIFT	Yes	8	ISA	W	●	●	●	●	●	Source	Simulation
HyperFlow [41]	2018	IFC	Yes	8	ISA	PT			●		●	None	FPGA
SDMP [90]	2018	Memory Safety	Yes	11	API	W	●	●	●		●	Source	Simulation
Typed Arch. [53]	2017	N/A	No	9	ISA	D	●	●	●	●	●	Source	FPGA
Dover [106]	2017	Programmable	Yes	Large	API	LT	●	●	●		●	Source	FPGA
Shakti-T [67]	2017	Memory Safety†	No	1	ISA	W	●	●	●	●	●	Source	FPGA
HDFI [99]	2016	DIFT	No	1	ISA	D	●	●	●	●	●	Source	FPGA
lowRISC [64, 101]	2015	Programmable	Yes	4	ISA	D	●	●	●	●	●	Source	FPGA
Taxi [43, 47]	2015	Memory Safety	No	8	Auto	D		●	●	●	●	Binary	Simulation
PUMP [30, 35, 37]	2014	Programmable	Yes	32	API	LT	●	●	●		●	Source	Simulation
CHERI [116, 117, 121]	2014	Programmable	No	1	Auto	W	●		●		●	Source	FPGA
SPARC M7 [75, 84, 88]	2014	Memory Safety	No	4	Auto	D		●	●	●	●	Binary	ASIC
Low-Fat Pointers [57]	2013	Memory Safety	No	8	Auto	W		●	●	●	●	None	FPGA
SAFE [5, 36]	2012	Programmable	Yes	59	API	W		●	●	●	●	None	FPGA
DataSafe [17]	2012	IFC	Yes	10	Auto	D			●		●	Binary	Simulation
Harmoni [33]	2012	DIFT	Yes	≤32	API	D		●	●	●	●	Binary	FPGA
Shioya, *et al.* [93]	2011	IFC†	Yes	12	API	PT		●	●	●	●	Binary	Simulation
SIFT [78]	2011	DIFT	Yes	32	API	D		●	●	●	●	Binary	Simulation
FlexCore [32]	2010	Programmable†	Yes	Large	API	D	●	●	●	●	●	Source	FPGA
Execution Leases [109]	2009	IFC	No	1	Auto	D	●	●	●	●	●	None	FPGA
GLIFT [110]	2009	IFC	No	1	Auto	D			●		●	None	FPGA
TIARA [94]	2009	Programmable	Yes	32	API	D			●		●	None	Theoretical
DIFT Coprocessor [52]	2009	DIFT†	Yes	4	API	D		●	●	●	●	Binary	FPGA
HardBound [34]	2008	Memory Safety*	No	1	Auto	D	●	●	●	●	●	Binary	Simulation
Loki [123]	2008	DIFT	No	32	Auto	PT		●	●	●	●	None	FPGA
FlexiTaint [113]	2008	DIFT	Yes	Small	API	D		●	●	●	●	Binary	Simulation
SECTAG [4]	2007	Programmable	Yes	Small	Auto	D	●	●	●	●	●	Source	Simulation
Raksha [26]	2007	DIFT	Yes	4	API	W		●	●	●	●	Binary	FPGA
SecureBit [86]	2006	DIFT	No	1	Auto	W		●	●	●	●	Binary	Emulation
Minos [25]	2004	DIFT	No	1	Auto	D		●	●	●	●	Binary	Emulation
DIFT [105]	2004	DIFT	No	1	Auto	PT		●	●	●	●	Binary	Emulation
RIFLE [111]	2004	IFC	Yes	Large	Auto	D		●	●	●	●	Binary	Simulation
AEGIS [104]	2003	IFC	No	Large	Auto	E			●	●		None	Simulation
Mondriaan [119, 120]	2002	Programmable	No	2	ISA	PT		●	●	●	●	Binary	Simulation
Aries [8]	2001	IFC	No	Large	Auto	W		●	●	●	●	None	Theoretical
XOM [61]	2000	IFC	No	Large	Auto	E			●			Binary	Simulation
M-Machine [15]	1994	IFC	No	2	Auto	W		●	●	●	●	None	ASIC
KCM [6]	1989	Memory	No	32	Auto	D			●		●	None	ASIC

An interesting corner case with non-configurable tags is CHERI [121]. CHERI's capabilities represent a fully programmable (within the capability paradigm) policy system, however the capability policy system is not reliant on the underlying tags. CHERI utilizes one bit tags to integrity protect its capabilities, and for no other purpose. Consequently, the tags are non-configurable despite CHERI's programmable policy interface.

Partially Configurable. Partially configurable architectures are user controlled, but only within their paradigm. For instance, IFC policies are all very similar, with most of the differences coming from how they are implemented in hardware. From a policy perspective, the key difference is how many flows can be tracked at once, with later implementations supporting four or more separate flows [26, 41, 52, 63, 78, 80, 111]. Consequently, flows from keyboard vs. network input can be distinguished, or multiple kinds of sensitive data tracked. Another distinguishing factor between IFC centric tagged architectures is whether they have fixed propagation rules, or present ISA extensions / control mechanisms that allow users to choose their own [26, 41, 63, 78, 80, 111].

Fully Configurable. Programmable tagged architectures support different programming paradigms, including user-specified tags [37] or generalized IFC [33]. PUMP [37] is an archetypal tagged architecture in this category. It provides infinite tags, whose semantics are entirely programmable within the domain specific language provided for tag policies. Unlike partially configurable architectures, there are no inherent limitations on how tags can be used, or the information they can store.

FlexCore [32] takes this even further and uses an FPGA as a policy co-processor, and reflashes it at boot with the desired tagged architecture. This allows it to mix the benefits of having a dedicated policy in silicon with the flexibility benefits of full programmability as it is infinitely re-configurable.

5.2 Tag Implementation

There are many design considerations for tagged architectures that affect size, weight, and power (SWaP) of the chip, including how to integrate tags into memory, where/when to do the extra tag-policy processing required, and how large to make the tags. Here we focus on implementation details that are noticeable from software, or have policy ramifications. In particular, we examine three implementation decisions: (i) whether tags are stored inline or disjoint, (ii) how tags are managed, and (iii) how large the tags are.

Tag Storage. Tags can be stored principally in one of two ways. In the first scheme, tags can manifest as register and cache extensions [6, 26, 34, 52, 67, 69, 69, 80, 80, 86, 97, 101, 105, 111, 118, 125], allowing them to be stored inline with the memory objects they reference, and flow through the memory hierarchy into registers. Doing so presents two challenges: (i) extensive modifications are required throughout the architecture, and (ii) tags are not isolated and are thus more vulnerable to attacker modification. The alternative, and currently dominant approach, is to store tags disjointly [4, 32, 33, 37, 47, 93, 106, 109, 110, 113, 121]. This separation usually happens at the register and L1 cache level, before being merged in L2. However, architectures with tag co-processors can offer completely separate tag-cache systems, though they are currently stored on the same physical memory. Given recent hardware-based attacks (*e.g.,* [55, 62]), fully separating tags (even within main memory) and putting them on a dedicated processor has become a security imperative.

Management. Tags can be managed in three ways: automatically by the hardware, explicitly via ISA extensions, or through a programmable API interface separate from the application. Automatic management is the most common case, particularly for architectures whose tags are not programmable. Recent programmable architectures, however, do not

rely on automatic tag management. ISA extensions [41, 53, 63, 67, 99, 101, 118, 120] give the programmer the explicit ability to interact with the tags and perform policy checks. In contrast, API management schemes [26, 32, 33, 37, 78, 80, 90, 93, 95, 105, 106, 113] have an interface for setting tag propagation and policy rules, for instance via a configuration register [80], software defined policies [37], or even changing FPGA configuration [32]. Whether the flexibility of having tags be explicitly manipulated by ISA extensions adds additional attack surface compared to API schemes, where the tags are typically only indirectly modified, is an open research question.

Tag Size. Tags come in three size classes: 1 bit, "few" bits, and unlimited bits. The second size class is deliberately ill defined; examples tend to cluster around 4-, 32-, or 64-bit tags depending on the targeted policy and when the architecture was developed. One bit architectures were prominent historically, and used for simple taint tracking IFC policies, as they had minimal cost in terms of extra storage requirements. With the advent of disjoint tag storage and modern fully programmable policies, the trend has been towards unlimited tags, or capabilities that can encode arbitrary amounts of information.

5.3 Trusted Computing Base

An important dimension to consider when examining tagged architectures is their Trusted Computing Bases (TCBs), *i.e.,* the set of components that the system assumes to be correct in order to provide its security guarantees. In this work, we focus on the TCB for enforcing policy with tags. A deployed system might have a larger TCB to ensure, for example, that the correct applications are loaded and run properly. TCB components for different architectures include the: compiler, processor, bootloader, operating system, and main memory. All tagged architectures include the processor in their TCB. We find three common patterns of inclusion in the TCB for other components. The first trusts the processor, bootloader, kernel, and main memory while the second adds the compiler to these four components. To compensate for the larger TCB, these systems are typically more programmable and offer the ability to enforce much richer policies. The final type trusts only the processor and memory and typically provides some variety of simple IFC policy.

Compiler. The compiler is included in the TCB for a tagged architecture if semantic information that can only be recovered from source code is required to enforce the tag policy. Tagged architectures that enforce DIFT or IFC policies, *e.g.,* Loki [123] and DIFT [105], typically do not require compiler support. Instead, they track information from a defined set of sources, *e.g.,* system calls, and monitor how such data is used to detect data leaks or control-flow hijacking. Other architectures require compiler support either because they introduce new instructions that manipulate and check tags [4, 53, 63, 80, 99, 117] or because they rely on the compiler to create the initial state of tags in the binary [37, 90, 106]. Systems in the first group rely on the compiler to correctly insert tag manipulation and checking instructions. Those in the second group rely on the compiler to generate an initial set of tags that will be applied to the binary on load. This enables policies about stack usage [90], control flow [37], and memory safety [30] that leverage information available to the compiler that is lost during compilation, trading off increased TCB for policy richness. Note that even if such policies rely on code annotations instead of static analysis the compiler would still be responsible for generating the correct tags from the annotations and so remain in the TCB.

Processor. One component in the TCB common to all tagged architectures is the processor. One element of the processor that deserves particular discussion is the Memory Management Unit (MMU), which provides simple read, write, and execute permissions on a page granularity. As such, it provides very similar, although much more limited, functionality

to a tagged architecture. Tagged architectures, therefore, have to make choices about whether and how to interact with the MMU. Most works on tagged architectures place tags on physical memory, enabling interaction with an MMU if one exists, but are otherwise ambivalent about the existence of an MMU at all [35, 47, 90, 99, 106]. However, some architectures such as CHERI [121], Low-Fat Pointers [57], and Mondriaan [119] intentionally combine the MMU with tags, leveraging the MMU for course-grained isolation or compatibility with existing software. For further comparisons between the memory protections afforded by modern MMUs, CHERI, and other schemes, we refer the reader to [121].

OS. The OS and system services, such as allocation, are also frequently trusted by tagged architectures. This may be because the OS must configure policy and mark information to be tracked [33, 36, 52, 57, 78, 88] or because the OS has the potential to arbitrarily manipulate the tag state and must be trusted not to do so [47, 53, 63, 93, 99, 101]. A good example of the first case is DIFT [105], which is designed to perform IFC to ensure that user input never directly influences control flow. To do so, it relies on the OS marking untrusted input. Other IFC/DIFT systems such as FlexiTaint [113] require the OS to configure tag propagation and checking. M7 [84, 88] represents a final example of this case. It implements a memory-safety policy by replacing `malloc` with a trusted allocator and recording metadata in the tags of allocations and pointers. TMDFI [63] is an example of the second case. Because tags must be checked and propagated explicitly with instructions, the kernel must be trusted. Architectures that trust the compiler also frequently end up in this case, due to a need to trust the OS loader to not modify initial tags or instructions needed to manipulate tags [53, 63, 80, 101, 118]. There are a few interesting exceptions, including Dover [106], PUMP [37], and CHERI [117, 121]. Dover and PUMP do not need to trust the OS because the bootloader actually loads the initial tags into the system along with the kernel; once started software cannot modify tags except as controlled by policy. This places the bootloader, but not the OS, in the TCB. CHERI is a capability based system that, among other things, can replace pointers with capabilities. Tags, however, are used merely to distinguish capabilities from normal memory, and this distinction is enforced by in the processor. As a result, CHERI does not require trusting the OS to correctly manipulate tags.

Bootloader. Many tagged architectures also require trusting the system bootloader. This is either because they include the OS in their TCB and rely on the bootloader to correctly start the trusted OS [33, 36, 47, 52, 53, 57, 63, 78, 88, 93, 99, 101] or because they rely on the bootloader to setup initial compiler-generated tags correctly [37, 90, 106]. This second case is the most interesting. In these systems, tags are inaccessible to a running system and are checked and manipulated by the processor according to a programmable policy. During boot, the tag policy and initial tag state is configured by the bootloader, placing the bootloader in the TCB.

Memory. A final element common to the TCB's of many tagged architectures is main memory. Most tagged architectures store tags in main memory and implicitly trust that tags will be stored and fetched from main memory correctly. The only two exceptions we know of are XOM [61] and AEGIS [104], both of which encrypt all data in main memory.

5.4 Compatibility

Tagged architectures and the policies they support leverage semantic information at different levels of the software stack. This in turn affects compatibility with existing software. Specifically, tagged architectures fall into three categories with respect to compatibility with existing code bases: no compatibility, binary compatibility, and source compatibility.

No Compatibility. Systems with no compatibility require manual source changes, either because they provide additional security controls beyond tags [41, 104, 123] or because they are theoretical clean-slate exercises [5, 36, 94, 109, 110]. HyperFlow [41] provides timing guarantees and IFC that goes beyond tags and requires source changes as compilers lack the semantic information to correctly use its ISA extensions to enforce these properties without developer input. TIARA [94] and SAFE [36] are examples of the second case. Both of these are clean-slate designs that develop radically different ISAs and may lack standard features such as virtual memory. The Gate Level Information-Flow Tracking (GLIFT) [109, 110] papers are also radical clean slate exercises that completely re-envision micro-architectures, and programming paradigms, in order to have fully precise information-flow tracking at the gate level. Consequently, even though realized in FPGA with custom microbenchmarks, they are effectively theoretical exercises.

Binary Compatibility. At the other end of the spectrum are systems that support existing binaries without modification or recompilation. These systems usually provide some form of IFC or DIFT [17, 25, 26, 47, 52, 78, 80, 86, 93, 105] or tag memory allocations and check accesses [34, 84, 88]. Existing binaries do not have to be modified for such systems because the information they require is a tractable binary static-analysis problem, *e.g.,* sources of user input for DIFT policies or heap allocations via `malloc` for bounds-checking policies. Note, however, that it is the required *analysis* not the *policy* that determines compatibility. For instance, Mondriaan [119] is a binary-compatible system that provides neither IFC nor bounds checking, but instead uses tags to provide traditional memory permissions (read, write, execute) at the word granularity for multiple users. Static binary rewriting is in scope for binary-compatible solutions [63, 99], though no prototype to date has demonstrated this.

Source Compatibility. Between these extremes exist a variety of systems that require recompilation to realize the protection enabled by the architecture. Unlike binary-compatible systems, which are limited by the information binary analysis can recover, these systems offer more flexibility, which allows a broader range of policies and designs. These systems typically introduce additional ISA instructions [4, 53, 63, 67, 99, 101, 118] or involve the compiler in the process of specifying the desired policy [37, 90, 106, 117]. HDFI [99] provides building blocks for programmers to construct the desired policy, and introduces new instructions to use these building blocks, thus requiring recompilation. PUMP [37] and CHERI [117] are notable examples of the second case – architectures that compile a programmer-defined policy into the binary. PUMP [37] implements a highly flexible tag engine that can compute arbitrary policies over instruction and operand tags. Semantic information from the source code is usually required to set the initial state of the tags for these policies. The CHERI [117] capability system includes a compiler that replaces all pointers with capabilities. A compatibility mode exists that only converts specified pointers to enable backward compatibility with existing binary code. The CHERI project has succeeded in recompiling the whole FreeBSD userspace with their system [28].

We have discussed compatibility so far in terms of what is required to realize the improved security offered by a tagged architecture. A related question is whether legacy binaries can be run with no protections. We believe that all of the systems we have classified as source-compatible (*i.e.,* requiring recompilation) support unmodified binaries at least for userspace programs, but those binaries will get no security benefits from the tagged architecture. This definitely includes TDMFI [63], SDMP [90], HDFI [99], Dover [106], PUMP [37], and CHERI [121].

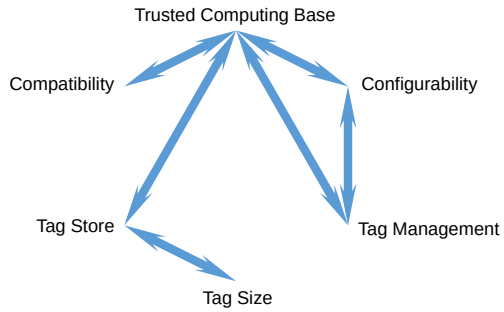

Fig. 4. Strongest trade-offs involved in tagged architecture designs.

Another important compatibility question is whether protected programs are compatible with unmodified libraries and vice versa. A few architectures successfully support this, notably HyperFlow [41], CHERI [117], and DataSafe [17], but most have not even considered it and do not appear likely to support it. See further discussion of this open challenge in Section 8.

6 DESIGN CHOICES AND TRADE-OFFS

In this section, we discuss the major design choices and trade-offs involved in a tagged architecture. We note that a tagged architecture is a complex system, and as such, every design choice should really be viewed holistically and it may impact all other dimensions of the design. Here, however, we emphasize some of the strongest trade-offs that we observe in the community to shine light on these choices. We also note that all of these design choices impact the overheads of a tagged architecture (silicon, power, runtime, and memory), which we discuss in-depth in Section 8, so we do not discuss them in this section.

Figure 4 illustrates the strongest tradeoffs in a tagged architecture design. An arrow denotes a strong trade-off whereas no arrow between two design choices denotes no dependence or weak dependence on some specific designs only.

The design choice that has the strongest impact on all other choices is the TCB. There is a tradeoff between the TCB and the compatibility of a design because if the design needs to be source compatible, it often uses the compiler to insert instructions for checking or manipulating the tags [4, 53, 63, 80, 99, 117], which puts the compiler in the TCB. On the other hand, binary compatible designs and those with no compatibility can avoid putting the compiler in the TCB [5, 36, 41, 43, 47, 57, 75, 84, 88].

The tag store and the TCB are also related. If the tags are stored in widened memory, disjoint memory, a page-table-like structure, or a lookup table structure (W, D, PT, or LT in Table 1, respectively), the memory is necessarily part of the TCB. On the other hand, if the tags are stored in encrypted memory (E in Table 1), the memory is not part of the TCB.

Tag management also impacts the size of the TCB. The logic that is used to manage tags (whether automatically, by ISA, or by API) is necessarily part of the TCB. This can be part of the kernel, the bootloader, and/or parts of the processor, depending on the implementation. The more complex this logic is, the larger that part of the TCB becomes.

Configurability also impacts the TCB although this impact is more design-specific. When the tagged architecture is non-configurable, the parts of the system that are used for

configuration purposes *could* be removed from the TCB although because of the other aspects of the design they may still be part of the TCB. In other words, when a tagged architecture is fully or partially configurable and relies on certain system parts (*e.g.,* compiler, bootloader, or kernel) for such configuration, those system parts are often included in the TCB, but when the architecture is non-configurable, those parts may or may not be part of the TCB depending on the oethr aspects of the design. For example, GLIFT [110] and XOM [61] are non-configurable and they do not have the compiler, the kernel, or the bootloader in their TCB, while HDFI [99] has all of them in its TCB despite being non-configurable due to the policy that it enforces.

Configurability is also related to tag management. Traditionally, tagged architectures that were non-configurable had automatic tag management [34, 57, 84, 105, 123] and those that are fully configurable either use ISA extensions or API management [41, 53, 63, 67, 99, 101, 118, 120]. This is primarily because non-configurable architectures enforce a fixed policy in hardware, so it is easy for them to implement the management logic automatically. In contrast, partially or fully configurable architectures need to expose some interface to the programmer to configure them (via ISA extensions or an API), so they can rarely be automated. Exceptions do exist when the policy is limited in scope (*e.g.,* enclave isolation in DataSafe [17]).

Finally, tag size impacts tag storage. Small tags that are used to distinguish special data (*e.g.,* capabilities or enclave designators) from other code and data in memory are often stored by widening the memory. CHERI [121], Timber-V [118], and M-Machine [15] are examples of such a design. Those that store metadata directly in tags, similar to HardBound [34], SPARC M7 [75, 84, 88], or Taxi [43, 47], often do so in disjoint memory. Additionally, those that have large, configurable tags, often store them in a disjoint memory region, a page-table-like structure, or a lookup table [30, 32, 35, 37, 106]. This is because small tags to distinguish special data are easier to fully handle within the main pipeline, while large configurable tags often require a complex policy engine to handle, so storing the tags disjointly or in separate tables may be desirable to implement such an engine.

7 EVALUATION

In evaluating tagged architectures, the focus is usually on the overhead introduced by the architecture and, for architectures that define a particular policy, the effectiveness of the policy being enforced. Historically, the evaluation of tagged architectures has focused on simulators and emulators due to the effort and cost involved in creating real hardware. However, over time there has been an increasing expectation of microarchitectural realism, driven by the reduced cost and ease of use of FPGAs, resulting in more realistic measurements. Unfortunately, it remains difficult to compare results obtained from different systems due to differences in policy, evaluation metrics, and benchmarks. One result of this is that few efforts attempt to compare to other tagged architectures. A summary of the evaluation methods for the tagged architectures we examine is shown in Figure 5. The rest of this section discusses the performance impacts of policy, performance metrics and benchmarks used, and the evaluation of other elements such as silicon area and security.

A major challenge in evaluating tagged architectures and comparing results is that the policy enforced by an architecture has a significant performance impact. PUMP [37] exemplifies this observation. They report 11% runtime overhead for tag maintenance with an additional 1% to 21% *average* overhead depending on the policy implemented. SDMP [90] reports a similar phenomenon, with runtime overhead varying between 1% and 11% *average* depending on policy, as does FlexiTaint [113]. For some configurable tagged architectures,

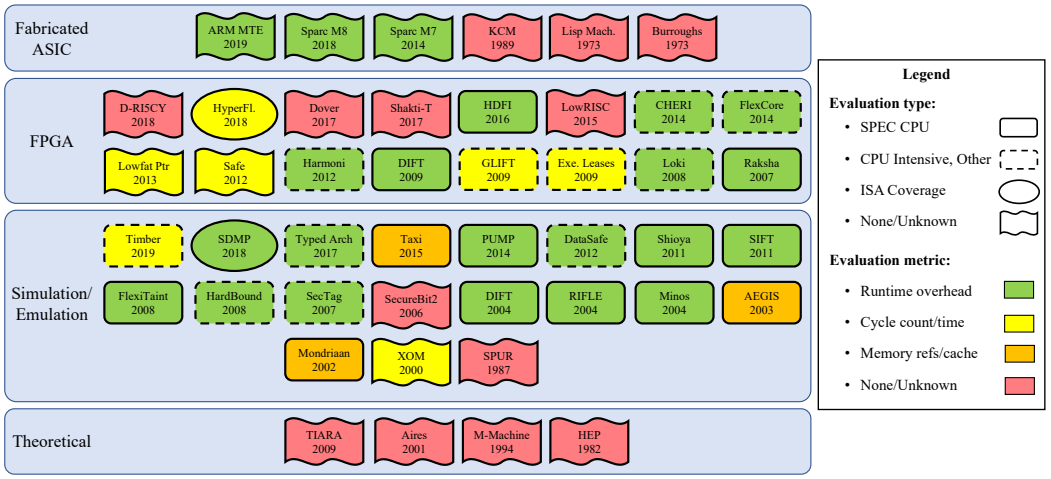

Fig. 5. The evaluation of tagged architectures in the literature.

only the tag-maintenance overhead is reported [41, 63, 93]. This challenge also impacts non-configurable architectures because their reported overhead implicitly includes their policy.

Performance metrics pose another challenge to comparing results. For an accurate comparison, we want to take into account as many overhead sources as possible. Runtime overhead, the difference in time taken to run a benchmark program on the tagged architecture vs. an equivalent untagged architecture, is good for that. It includes additional instructions or cycles as well as cache behavior and OS overheads. However, several efforts consider only the additional cycles introduced [41, 63], the cache miss rate [53, 90], or increases in cycle time [36, 57, 61]. While these metrics provide useful information, they frequently omit important overheads. For example, the Low-Fat architecture [57] measures the increase in cycle time, which does not take into account software execution and does not include any cache overhead resulting from the introduction of tags. We encourage the use of runtime overhead as the primary metric, especially on FPGA or ASIC implementations, which have good microarchitectural realism.

While runtime performance is a key consideration for tagged architectures, it is not the only one. In particular, the silicon overhead of a tagged architectures is also particularly relevant. This overhead measures the silicon area devoted to tag processing that could otherwise be used for other architectural features such as larger caches. A number of efforts consider the silicon area used by their tag processing, including Typed Architectures [41], CHERI [121], SAFE [36], Harmoni [33], and others. Again, these overheads vary dramatically based on the architecture and complexity of the base core. Along similar lines, a few architectures [52, 78, 113] dedicate an additional core to tag processing. This essentially reduces the number of cores available to process application data and should be considered when evaluating such architectures.

Architectures that define and implement a specific policy frequently include a security evaluation. This often takes the form of a benchmark suite of exploits [80, 105], manually developed exploits [43, 47], or known exploits in existing software [25, 26], which are demonstrated to fail.

One notable evaluation effort is the CHERI project, which has published a paper exploring the performance tradeoffs and overheads of implementing single-bit tagged memories [51]. This paper covers different ways to implement single-bit tags, integrate them into the memory hierarchy, and the caching behavior of these tags for different size caches.

8 CHALLENGES

In this work, we have identified several challenges to the design, development, and adoption of tagged architectures that remain open. We group these challenges into three broad categories: (i) provisioning, (ii) enforcement, and (iii) overheads, based on the life-cycle of a tagged program, as illustrated in Figure 1.

The life-cycle of a tagged program involves two major elements: provisioning and enforcement. Provisioning involves generating correct tags for a program and then securely storing them until the program is executed. Enforcement involves executing the program and using tags to apply one or more policies on that execution. Additionally, policy enforcement via tags results in a variety of overheads from memory to silicon area. We identify challenges in each of these areas and discuss them below.

Table 2 summarizes the treatment of the challenges discussed in this section by the various tagged architectures we have examined in this work. Red in the table denotes little to no treatment of the challenge. Yellow denotes some consideration of the challenge. Green denotes a complete addressing of the challenge. White indicates a challenge that is not applicable to the given architecture. As can be observed, most challenges have not received sufficient treatment in the existing tagged architectures.

8.1 Provisioning

Before tagged hardware can enforce policy, the initial tags for applications and the tag-propagation logic must be created. We refer to this process of generating and storing tags as provisioning, and it is a critical, but often overlooked, part of any practical tagged architecture. If the initial program tags are tampered with, then the tag policies are compromised. Note, however, that this is not different from ensuring that any binary, tagged or otherwise, is not modified before execution and can be dealt with by, *e.g.,* signing the binary and tags.

AEGIS [104] is noteworthy for bypassing this issue altogether by generating and propagating tags in the processor and encrypting tags in off-chip memory. DataSafe [17] takes a different approach and uses a privileged hypervisor to validate the authenticity of policies transferred from a remote machine using encryption. Both designs ensure authentic tags by leveraging isolation and encryption to validate the authenticity of tags or the policies from which they are generated.

Even with authentic tags, properly tagging and checking the tags on a program is a complex task, and it is possible for the tag and check insertion code to be buggy, allowing policy violations. More concretely, suppose the initial tag state is generated by a compiler, as in PUMP [37]. The compiler must now correctly generate the initial tags, as well as the binary for execution. Such concerns have been around at least as long as compilers have [108]. Note that a similar situation exists whether the initial tag state is provided by the compiler, loader, or OS. In all cases, potentially buggy code must be relied upon to process every binary and generate correct tags. We call this the tag-correctness challenge.

One possible method to ensure tag correctness is formal verification of the compiler. While formally verified compilers exist for some languages, such as a subset of C [60], developing such compilers is a challenging task and such compilers often lack the optimizations expected from a production compiler. Another option is to exhaustively test the program to verify

Table 2. Evaluation of how each tagged architecture addresses the challenges we outline. Red – little to no consideration. Yellow – some consideration. Green – completely addressed. White – not applicable.

Name	Year	Prov	Enforcement							Overheads			
		Tag Correctness	Lang Ambiguity	Policy Limitations	Policy Composition	Dyn Link/Load	DMA	Advanced HW	Side-Channels	Memory	Performance	Silicon Area	Power
Timber [118]	2019												
ARM MTE [3]	2019												
D-RI5CY [80]	2018												
TMDFI [63]	2018												
HyperFlow [41]	2018												
SDMP [90]	2018												
Typed Architectures [53]	2017												
Dover [106]	2017												
Shakti-T [67]	2017												
HDFI [99]	2016												
lowRISC [101]	2015												
Taxi [47]	2015												
PUMP [37]	2014												
CHERI [121]	2014												
SPARC M7/M8 SSM [84]	2014												
Low-Fat Pointers [57]	2013												
SAFE [36]	2012												
DataSafe [17]	2012												
Harmoni [33]	2012												
Shioya, *et al.* [93]	2011												
SIFT [78]	2011												
FlexCore [32]	2010												
Execution Leases [109]	2009												
GLIFT [110]	2009												
TIARA [94]	2009												
DIFT Coprocessor [52]	2009												
HardBound [34]	2008												
Loki [123]	2008												
FlexiTaint [113]	2008												
SECTAG [4]	2007												
Raksha [26]	2007												
SecureBit [86]	2006												
Minos [25]	2004												
DIFT [105]	2004												
RIFLE [111]	2004												
AEGIS [104]	2003												
Mondriaan [119]	2002												
Aries [8]	2001												
XOM [61]	2000												
M-Machine [15]	1994												
KCM [6]	1989												
SPUR [125]	1987												
Lisp Machine [69]	1985												
HEP [97]	1982												
Burroughs [76]	1973												

```
1  // Start as integer for pointer arithnetic.
2  uintptr_t p = 0xdeadbeef;
3  ...
4  // Up-cast from integer to pointer type.
5  uintptr_t *q = (uintptr_t *) p;
6  ...
7  // Policy violation despite being expected
8  // behavior (false positive in some designs).
9  *q = 1;
```

Fig. 6. Upcasting is an ambiguous idiom that causes problems for memory-safety policies.

that tags behave as expected on all inputs. Unfortunately, exhaustively generating all inputs for most reasonable programs is impractical, though fuzzing is gaining in popularity as an approximation of this approach. A final option would be to leverage symbolic execution to identify and explore all paths that may lead to tag violations [13]. Unfortunately, this approach does not scale due to a path explosion problem, as the number of paths grows exponentially with program size [7, 11]. In short, guaranteeing tag correctness is still an open problem, and equivalent to guaranteeing the correctness of any piece of software.

So far we have considered the correctness of *tags*, and not the policies enforced on the semantics of those tags. As illustrated in subsection 2.2, writing a policy that enforces the intended properties is non-trivial. Tags provide a powerful security mechanism, but they are still subject to the usual problems of correct specification and implementation of software.

8.2 Enforcement

Policy enforcement consists of maintaining, propagating, and checking tags during program execution. This is the core of any tagged architecture and performing this computation efficiently has been the focus of much existing work. However, there are several areas that have not been well explored, particularly around ambiguous or limited policies, the composition of policies, and advanced hardware features. We discuss these issues and the complications they introduce below.

8.2.1 Language Ambiguity. A variety of efforts we examined have reported false positives and false negatives that occurred during the testing of common policies, especially those related to memory safety [20, 43, 77]. In other words, these architectures incorrectly reported violations in benign program execution or missed some execution that violated policy. One of the major reasons for these false positives and false negatives is language ambiguity that makes determining whether an operation is a policy violation difficult or impossible. The C programming language is particularly prone to these issues, especially for memory safety policies [20, 68].

One problematic idiom common in C is upcasting (*e.g.,* an integer to pointer cast), represented in Figure 6. Here, an integer is directly cast to a pointer, an unavoidable operation in systems programming, especially when interfacing with hardware. Should a memory-safety policy allow such a cast, and if so what should the resulting pointer be allowed to access? As the operation is required for, *e.g.,* OS code, policies must have an answer. Assuming it is allowed, the simplest default access permissions are everything or nothing. A sufficiently complex analysis might be able to track pointers across casts (in the case of a pointer-to-integer-to-pointer sequence) or to identify the memory object being pointed to based on the pointer's value. In practice however, existing tagged architectures give all or nothing permissions. Hardbound [34], for instance, will set the bounds for this newly created

```
1   struct {
2     char buf[64];
3     int i;
4   } obj_t;
5
6   // Initialize obj.
7   obj_t obj;
8   ...
9
10  /* Part A */
11  void *op = (void *) obj;
12  ...
13  op[64] = 1; //legal access
14
15  /* Part B */
16  op = (void *) obj.buf;
17  ...
18  op[64] = 1; //also legal
```

Fig. 7. C allows the first field of a struct to be used as a reference for the struct itself.

pointer to `NULL`, resulting in a policy violation when the pointer is dereferenced. If there is memory allocated at this address, this will be a false positive.

Another problematic idiom is C's conflation of the first element of a struct with the struct itself, as shown in Figure 7. In this code, a pointer to a struct is used to access the second field, which is legal. Then a pointer to the first field of the struct is used to access the second field. This is also legal because C defines a pointer to a struct and a pointer to the struct's first field to be equivalent [45]. However, this second access is not in the spirit of memory safety and many systems, including Hardbound, disallow it [34]. Note that unlike upcasting, which is a result of undefined behavior, this situation is defined behavior, but ambiguous from a memory-safety perspective [45].

For ambiguous cases, there is a tradeoff between strictly enforcing policy and false positives. Some architectures provide flexibility to toggle between such choices, while others do not.

8.2.2 Policy Limitations. While tagged hardware can enforce a variety of policies, it is sometimes necessary to make tradeoffs between the ideal policy and what is actually enforced, usually to match the hardware capabilities or for performance reasons. This usually introduces false positives or false negatives to the policy-enforcement process.

There are application-specific policy simplifications that can be made. For instance, Taxi [43, 47] defines a "linearity of return address" policy in which only one copy of a return address can have a return-address tag (*i.e.,* when a return address is copied, the destination strips the return address tag of the source). This policy represents an attempt to approximate a Shadow Stack while working within the hardware limitations. However, this creates a false positive when running the `gcc` torture tests because a return address that has already been used to call a function is used again. This is a case where policy approximation results in false positives.

Another reason that policy may be approximated is due to hardware limitations. For instance most of the memory-safety policies discussed in this paper are non-rigorous and incomplete, including Taxi [43], HDFI [99], HardBound [34], SecureBit [86], Minos [25], and DIFT [105]. This is due to limitations on the size of the tags supported by hardware. For instance, SPARC M7 [84, 88] uses a 4-bit tag to protect heap objects from overwriting each other's memory. Of course, many applications require more than $2^4 = 16$ unique memory allocations, leading to repeated tags. As a result, an attacker has a non-negligible probability

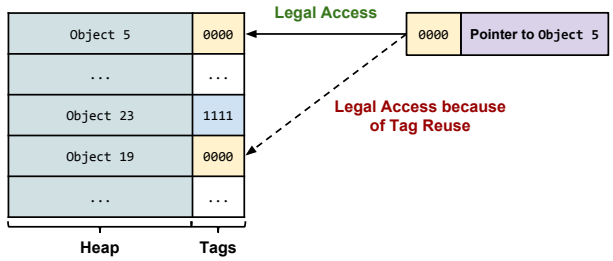

Fig. 8. SPARC M7 false negative caused by limited tag size.

of using a pointer to one heap allocation to interfere with the memory contents of another allocation with the same tag. A more savvy attacker can also target the $\frac{1}{16}$ of objects that share the same tag to hijack control (assuming that tag assignment is deterministic). This attack, which is demonstrated in Figure 8, leads to an undetected violation (*i.e.,* a false negative).

8.2.3 Policy Composition. Existing tagged architectures have demonstrated that tags can be used to implement a variety of policies including identifying and tracking pointers for memory safety [34], enforcing temporal memory safety [57], tracking type information [53], securing the stack and its return addresses [47, 90], enforcing control-flow integrity [37], taint tacking [37], tracking the flow of user input [86, 105], generalized information-flow tracking policies [17, 99], and the isolation of important metadata [117]. Many applications could benefit from several of these policies. For example, one might want to apply a policy that provides memory safety and a policy that tracks information flow to ensure that sensitive encryption keys are not leaked to disk or the network.

Unfortunately, most tagged architectures do not consider multiple policies being enforced concurrently. To see why this is challenging, consider a machine with a one-bit tag trying to enforce two policies: a memory-safety policy tagging return addresses with a 1 (and all other words with a 0) and a taint-tracking policy that tracks all external inputs with a tag of 1. When these policies are combined, the memory-safety policy is ineffective because external inputs are allowed to overwrite return addresses (since they both have a tag of 1) while the taint-tracking policy believes that all return addresses are derived from external inputs.

One way to compose policies is to use larger tags that can be segmented into sub-tags used for each policy being enforced. However, the larger tags required by this approach incur higher memory, silicon, and power overheads. A few architectures support combining multiple DIFT policies using this approach. Both the DIFT Coprocessor [52] and Raksha [26] provide a small, multi-bit tag where each bit tracks information flow for a different policy. However, even these systems only support enforcing a limited number of DIFT policies simultaneously; they do not support either arbitrary policies or arbitrary numbers of policies.

Dover [106] and PUMP [35] use a different approach for policy composition in which the tag itself is a pointer to metadata of potentially unbounded size. In this case, supporting multiple policies still requires more memory for tag metadata, but does not directly impact the size of the tags and associated cache behavior. Memory, silicon, and power overheads for the additional memory consumed by this metadata is still a concern, however.

8.2.4 Dynamic Linking and Loading. Dynamic linking enables applications to share the same external dependencies (*e.g.,* the C standard library) and reduces the duplication of code

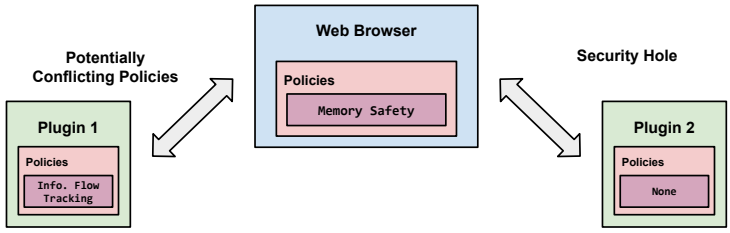

Fig. 9. Dynamic loading can lead to inconsistencies in policies as well as create attack surfaces in an application.

between processes. Dynamic loading allows applications to load and run new code modules, such as plugins and extensions, as needed at runtime. Both of these techniques can result in code compiled separately, with potentially different policy, interacting in the same process at runtime.

To see why this can be a problem, consider a shared library compiled without a tag policy dynamically linked with an application compiled with a tag policy. The application's policy cannot protect the library's code, leaving the library open to attackers. Worse, the library may actually corrupt tagging state used by the application to enforce its policy. Consider an application with a DIFT policy that tracks all external inputs into the program with a tag of 1. If our shared library (with no tag policy) copies user input without copying the tag, then this (tag-free) copy of user input can later be used by the application to violate policy (since it is not tagged as external input). This suggests that all code in a process must be compiled with the same tag policy in order for that policy to be effective.

When multiple applications have different tag policies, the situation gets even worse. Consider a tagged machine with a one-bit tag and two applications, A and B, sharing a library with a function `process`. Application A enforces a memory-safety policy by tagging return addresses with a 1 (and all other words with a 0). This policy rejects a return address with a 0 tag, preventing an attacker from hijacking the program's execution. Application B enforces a DIFT policy that tracks all external inputs into the program with a tag of 1.

Now suppose application B and its external dependencies are compiled and executed on the machine. This will result in the library function `process` being compiled with a DIFT policy. As a result, when application A is compiled and run, it must compile its own version of the `process` code. Otherwise, application A will run with a DIFT policy in the `process` function, allowing an attacker to use program input (tagged with a 1) to overwrite the return address of the `process` method and hijack A's execution.

Similarly, suppose a tagged machine runs a web browser. The web browser is compiled with a complete memory-safety policy (Figure 9). At runtime, the browser uses the Linux `dlopen` interface to dynamically load plugins that extend the functionality of the browser. There is no guarantee, however, that the plugins are compiled with the same policies as the browser, opening the application to the same challenges seen with dynamic linking. In essence, applications cannot leverage dynamic loading or linking without risking conflicting policies that open them up to attacks.

A few architectures side step this challenge and successfully support shared libraries, notably HyperFlow [41], CHERI [117], and DataSafe [17]. This is possible because these architectures use tags to implement only classes of policies that cannot conflict with each

other, like isolating capabilities from other elements of memory or only supporting IFC policies.

A more general approach would be to use large tags, such that any policy desired has its own sub-tag. However, large tags come with significant overhead in terms of memory, silicon, and power, as mentioned previously. Furthermore, a number of the architectures described in this work only provide one-bit tags [25, 86, 93, 99].

8.2.5 Direct Memory Access. Direct Memory Access (DMA) allows peripherals to copy data directly into memory, bypassing the processor. It is frequently used by high-bandwidth I/O devices such as network cards and disk drives to copy data directly to or from memory without involving the CPU.

The ability for other devices to read or write memory independent of the processor has significant implications for tagged architectures. Unless the DMA engine is made tag-aware, it completely bypasses any tag policy. Thus, tags cannot be used to prevent the DMA engine from reading or writing memory nor will the DMA engine update the tags after writing to memory. As a result, any isolation enforced with tags can be bypassed using the DMA engine and any tags on objects written by the DMA engine are not guaranteed to be correct.

Not only can a tag-oblivious DMA engine bypass and not update tags, it may also be able to rewrite the tags of arbitrary memory if tags themselves are stored in DMA-accessible memory. This is of particular concern due to the rise of DMA-based attacks that either do not require a malicious peripheral [91] or function despite the presence of an IOMMU [65]. The only architectures protecting against such DMA attacks (AEGIS [104], DataSafe [17], and XOM [61]) do so by encrypting all data in main memory, preventing the DMA engine from modifying the data or tags.

We argue that a practical modern tagged architecture must include a tag-aware DMA engine. The only works we are aware of considering DMA for tagged memory are WHISK [87] and TaintHLS [85] which investigate the integration of peripherals and DMA with tagged memory in SoCs. As none of the tagged architectures we examine in this work consider such integration, significant research into this area is still needed.

8.2.6 Advanced Hardware Features. Existing work on tagged architectures has been mainly confined to single cores with in-order, single-issue execution. However, modern general-purpose processors are significantly more complex with features such as multicore, out-of-order execution, and speculation that significantly improve performance. All of these features pose challenges to tagged architectures.

Out-of-order execution is a technique in which instructions that are independent of one another are re-ordered based on available execution units and input data. This results in a significant performance improvement by maximizing the utilization of processing elements. Speculative execution builds on out-of-order execution by making predictions about instructions or data to be executed after branches and preemptively performing those calculations. This expands the performance improvements that out-of-order execution provides across branches. Both have recently been shown to result in powerful side channels [55, 62].

Out-of-order and speculative execution significantly complicate tagged architectures. For out-of-order execution, tags and tag policy introduce additional constraints on when instructions are independent and can be safely re-ordered while for speculative execution, tags and tag policy introduce constraints on whether speculatively executed code can be committed. Consider an instruction A that updates the tag on the program counter to be x, an instruction B that writes to a memory address b, and a tag policy that only allows writes to b if the program counter has the tag x. While instructions A and B appear independent

of each other, the successful execution of instruction B *given this tag policy* depends on instruction A. In other words, determining whether two instructions are independent and can be executed out-of-order requires analyzing the tag policy.

The vast majority of tagged architectures we examined are based on in-order processors and do not consider out-of-order or speculative execution. The few architectures that consider out-of-order execution are largely information-flow tracking systems in which tags only change to follow propagating data [52, 78, 113] or fine-grained memory-protection schemes in which tags do not change [4, 119]. Similarly, the few systems that speculate are focused on basic memory-protection schemes [104, 119] in which tags do not change frequently. Dealing with out-of-order and speculative execution in the general case is an open problem.

Multicore is another performance-enhancing feature in modern processors, but it introduces significant complexity for tagged architectures that has not been yet studied. In such architectures, memory and often at least one cache level is shared among multiple cores. While it is conceptually possible to consolidate all tag processing in a central tag engine, this would likely impose an undue performance bottleneck by serializing all tag processing. This necessitates core-local tag processing. However, shared memory and caches introduce the possibility of tag-specific concurrency challenges, especially for shared-memory applications. Tags must be coherent in any caches to ensure correct policy enforcement. For parallel application processing, at minimum the policies enforced by each core must be consistent (or at least compatible) when operating on shared data. Some tag policies themselves may need to be updated to be concurrency safe. Furthermore, tag and memory accesses must be either atomic or properly synchronized, which may introduce performance overhead or undue complexity. These are only a few of the myriad challenges in designing a multicore tagged architecture that have been largely unexplored.

8.2.7 Side Channels. While tagged architectures enable the enforcement of many classes of security policies, they do not protect against all classes of attacks. In particular, tagged architectures by themselves do not protect against side-channel attacks. In fact, tagged architectures are likely to introduce additional side channels due to the additional caching and data-dependent computation inherent in tag processing. For example, the rule cache present in many architectures, including Dover [106] and PUMP [35], can likely be subjected to prime-and-probe style attacks to identify the tags used by an isolated software component or to determine its computation pattern. Similarly, the delay inherent in a tag cache miss is likely measurable and exploitable via a flush-and-reload or prime-and-probe attack.

Recent hardware-based side-channel attacks [55, 62] have highlighted the importance of microarchitectural side channels. Tagged processors are open to the same types of microarchitecture side-channels and possibly others related to additional tag complexity.

The only work we are aware of investigating this issue is HyperFlow [41], which is a processor that is designed to be both tagged and free of timing side channels. Hyperflow implements a nonmalleable IFC policy using tags. To eliminate timing side channels, the processor tracks the tag of the currently executing code and flushes caches, TLB, branch predictor, and other microarchitectural state on changes in the confidentiality or integrity tag of the running code. The modifications to avoid timing side channels seem more extensive than those to add tags. The authors report overheads in cycles per instruction of between 1% and 69%, largely due to padding the multiply operation to the worst-case number of cycles. While HyperFlow is a promising start, significant additional work is needed to understand side channels present in tagged architectures and suggest methods for remediation.

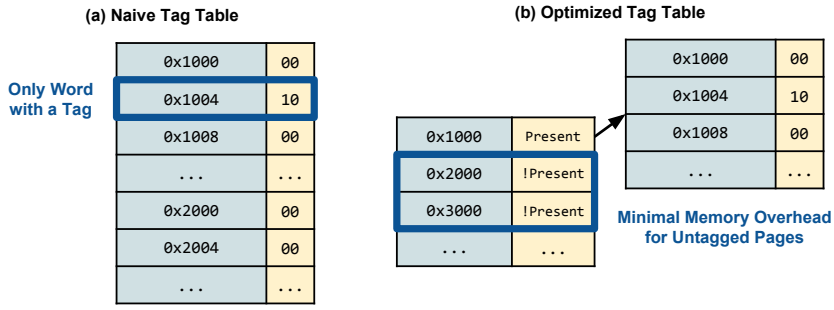

Fig. 10. A naïve and optimized implementation of an in-memory tag table.

8.3 Overheads

Tagged architectures add complexity that incurs several types of overhead. Understanding, characterizing, and reducing such overheads is crucial to the adoption of tagged architectures. We break down these overheads along four dimensions: memory overhead, runtime overhead, silicon overhead, and power overhead. Note that we do not consider storage overhead since filesystem storage is not tagged in any of the architectures described in this paper. Where tags need to be preserved beyond memory, *e.g.,* for tagged executables, a compressed representation is typically used and the tags are then reconstructed when loading into memory, resulting in comparatively little extra space being required.

8.3.1 Memory Overhead. The memory overhead of a tagged architecture depends on the size of the tags and any compression schemes used by the hardware. This presents several trade-offs as large tags enable more complex policies but have larger memory overhead. Similarly, complex compression schemes may reduce the memory overhead but increase the computational overhead of accessing the tags.

Most tagged architectures address the memory overhead of tags. However, memory overheads still vary significantly, due to the trade-offs mentioned above. Memory overheads can easily reach 100% for architectures with large, word-size tags that do not implement compression schemes, such as HardBound [34], while architectures with 1-bit tags and complex, page-table-inspired compression schemes, like Shioya, *et al.* [93], can achieve memory overheads as low as 0.685%. These schemes are illustrated in Figure 10.

8.3.2 Runtime Overhead. Runtime overhead is the slowdown on a tagged architecture relative to an untagged architecture. Two key factors contributing to this overhead are additional tag processing and increased cache pressure due to tags.

In terms of cache pressure, architectures that store tags disjointly from data in the last-level cache experience increased cache miss penalties. As shown in Figure 11, cache misses increase when both tags and data/instructions share the same cache. Similarly, if tags are stored disjointly from data in memory, then multiple fetches may be required on a single cache miss, one to fetch the data and one to fetch the tag. Even if memory is widened, there is still a runtime impact because an equivalently sized cache holds less data, due to the space used by tags. The CHERI team examined the caching and storage impact of their 1-bit tags in a recent paper [51], but more work is needed in this area.

Propagating and checking tags also introduces additional computation. For instance, several tagged architectures introduce instructions that the compiler can insert to check or

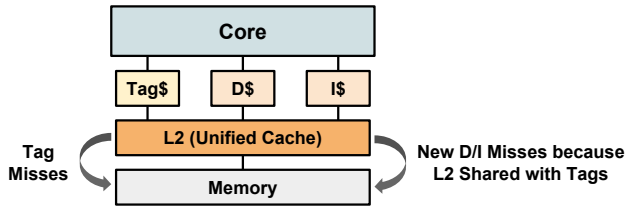

Fig. 11. Tagged architectures that share a cache between tags and data/instructions increase the number of cache misses.

update tags [34, 41, 99]. These instructions must be inserted anywhere that tag processing is required, increasing the runtime of the program. To eliminate these additional instructions, some architectures process tags in dedicated units in parallel with the ALU [37, 47, 106]. This approach, however, increases the silicon overhead of the architecture.

8.3.3 Silicon Overhead. Silicon overhead represents area on the processor that is dedicated to tag-processing logic and that would otherwise be available for additional cores, ALUs, or cache in an untagged architecture. The silicon overhead of a particular architecture is driven by the tag-processing mechanisms and its integration into the processor design.

Hardbound [34] and Minos [25] process tags using minor modifications to the existing pipelines and several new instructions, resulting in minimal additional area. Other architectures such as Taxi [43] and TIARA [95] add a dedicated tag-processing unit to process tags in parallel with instructions. On Dover, this tag processing unit is a full CPU [106]. These designs use significantly more area.

Policy and tag complexity also affect the silicon overhead. For instance, SecureBit [86] uses a 1-bit tag to protect buffers shared among processes. With OS modifications and a few new instructions, SecureBit implements a memory-safety policy with little silicon overhead. In contrast, PUMP [37] supports software-defined policies in hardware, requiring an additional pipeline stage with a dedicated tag-processing unit to fetch policies and interpret policies against incoming tags, all of which requires significantly more area. PUMP thus provides more flexible policies at the cost of more area. Tag size results in a similar trade off because the larger tags must be stored in caches, requiring more bits and thus more area for a cache that holds the same amount of data.

Additional silicon also impacts the TCB of the tagged architecture. Returning to the comparison above, SecureBit relies on the OS to support tag policies. In contrast, PUMP utilizes more silicon to provide a privileged processor mode to handle tag creation and propagation, isolated from a potentially untrusted OS. In summary, architectures tradeoff silicon in varying quantities for performance, policy complexity, and improved security guarantees.

8.3.4 Power Overhead. The additional silicon required by tagged architectures results in an increase in required power over an untagged architecture. Consider, for instance, a design such as HardBound [34], which can easily have 100% memory overhead. This can increase the amount of power required for DRAM due to the additional memory used for maintaining tags. Similarly, coprocessors, such as those used by Dover [106] and PUMP [35], are effectively processors themselves, resulting in nontrivial power overheads. This reality is at odds with

the SWaP (size, weight, and power) requirements of modern mobile and embedded devices in which power must be conserved in order to preserve battery [82].

Only five of the efforts discussed in this paper (Typed Architectures [53], PUMP [35], Harmoni [33], FlexCore [32], and FlexiTaint [113]) assess power overhead in their work. This absence and the importance of power/energy analysis for tagged architectures suggests significant room for future work.

9 CONCLUSION

In this work, we presented a survey of tagged architectures and developed a four-dimensional taxonomy to study the efforts in this area. We identified and studied a number of challenges in three broad categories of provisioning, enforcement, and overhead and found that existing tagged architectures provide little work to address them. We also discuss why addressing these challenges is non-trivial and pose a number of open research problems to advance the security and practicality of tagged architectures.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 340–353.

[2] arm. 2020. ARM Architecture Reference ManualARMv8, for ARMv8-A architecture profile. https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf

[3] arm. 2020. Memory Tagging Extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf?revision=ef3521b9-322c-4536-a800-5ee35a0e7665&la=en&hash=D510ED84099D3B8AA34723AC110D48E3A28FA8D6

[4] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K Jha. 2007. Architectural support for run-time validation of program data properties. *IEEE Transactions on very large scale integration (VLSI) systems* 15, 5 (2007), 546–559.

[5] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C Pierce, Randy Pollack, and Andrew Tolmach. 2014. A verified information-flow architecture. *ACM SIGPLAN Notices* 49, 1 (2014), 165–178.

[6] Hans Benker, Jean-Michel Beacco, M Dorochevsky, Th Jeffré, A Pöhlmann, J Noye, B Poterie, JC Syre, O Thibault, and G Watzlawik. 1989. KCM: A knowledge crunching machine. *ACM SIGARCH Computer Architecture News* 17, 3 (1989), 186–194.

[7] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 351–366.

[8] Jeremy Brown and Thomas F Knight Jr. 2001. A minimal trusted computing base for dynamically ensuring secure information flow. *Project Aries TM-015 (November 2001)* 37 (2001).

[9] Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi. 2013. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 131–146.

[10] A. Burks, H. Goldstein, and J. Von Neumann. 1946. Preliminary discussion of the logical design of an electronic computing instrument. Report to the U.S. Army Ordinance Department. Also appears in Ccomputer structures: Readings and examples., McGrawHill Inc., 1971, 92-120.

[11] Jacob Burnim and Koushik Sen. 2008. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering*. IEEE Computer Society, 443–446.

[12] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* (2017).

[13] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: Three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[14] Nicholas Carlini and David Wagner. 2014. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*.

[15] Nicholas P Carter, Stephen W Keckler, and William J Dally. 1994. Hardware support for fast capability-based addressing. In *ACM SIGPLAN Notices*.

[16] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 2005. Defeating memory corruption attacks via pointer taintedness detection. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*. IEEE, 378–387.

[17] Yu-Yuan Chen, Pramod A Jamkhedkar, and Ruby B Lee. 2012. A software-hardware architecture for self-protecting data. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 14–27.

[18] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, Huijie DENG, et al. 2014. ROPecker: A generic and practical approach for defending against ROP attack. In *NDSS*.

[19] CHERI. 2020. ARM Morello Board. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html

[20] David Chisnall, Colin Rothwell, Robert NM Watson, Jonathan Woodruff, Munraj Vadera, Simon W Moore, Michael Roe, Brooks Davis, and Peter G Neumann. 2015. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *ACM SIGPLAN Notices*.

[21] David Chisnall, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, Robert N.M. Watson, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, and Robert Norton. 2017. CHERI JNI: Sinking the Java Security Model into the C. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, 569–583.

[22] Victor Costan and Srinivas Devadas. 2016. *Intel SGX explained*. Technical Report. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086.

[23] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*.

[24] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. 1999. Protecting systems from stack smashing attacks with StackGuard. In *Linux Expo*. Citeseer.

[25] Jedidiah R Crandall, S Felix Wu, and Frederic T Chong. 2006. Minos: Architectural support for protecting control data. *ACM Transactions on Architecture and Code Optimization (TACO)* 3, 4 (2006), 359–389.

[26] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A flexible information flow architecture for software security. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 482–493.

[27] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2010. Tainting is not pointless. *ACM SIGOPS Operating Systems Review* (2010).

[28] Brooks Davis, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Robert N. M. Watson, Stacey Son, Jonathan Woodruff, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, and Nathaniel Wesley Filardo. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, 379–393. https://doi.org/10.1145/3297858.3304042

[29] Arthur Azevedo De Amorim, Maxime Dénes, Nick Giannarakis, Catalin Hritcu, Benjamin C Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-policies: Formally verified, tag-based security monitors. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 813–830.

[30] Arthur Azevedo De Amorim, Maxime Dénes, Nick Giannarakis, Catalin Hritcu, Benjamin C Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-policies: Formally verified, tag-based security monitors. In *IEEE Symposium on Security and Privacy*. IEEE, 813–830.

[31] Arthur Azevedo de Amorim, Cătălin Hriţcu, and Benjamin C Pierce. 2018. The meaning of memory safety. In *International Conference on Principles of Security and Trust*.

[32] Daniel Y Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G Edward Suh. 2010. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *MICRO '10*.

[33] Daniel Y Deng and G Edward Suh. 2012. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 1–12.

[34] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. 2008. Hardbound: Architectural support for spatial safety of the C programming language. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 103–114.

[35] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and Andre DeHon. 2015. Architectural support for software-defined metadata processing. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 487–502.

[36] Udit Dhawan, Albert Kwon, Edin Kadric, Catalin Hritcu, Benjamin C Pierce, Jonathan M Smith, André DeHon, Gregory Malecha, Greg Morrisett, Thomas F Knight, et al. 2012. Hardware support for safety interlocks and introspection. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2012 IEEE Sixth International Conference on*. IEEE, 1–8.

[37] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and André DeHon. 2014. Pump: a programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 8.

[38] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the point (er): On the effectiveness of code pointer integrity. In *SP '15*.

[39] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 901–913.

[40] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the Effectiveness of Type-based Control Flow Integrity. In *Proceedings of IEEE Annual Computer Security Applications Conference (ACSAC'18)*.

[41] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. 2018. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM Press, 1583–1600.

[42] Edward A Feustel. 1973. On the advantages of tagged architecture. *IEEE Trans. Comput.* 100, 7 (1973), 644–656.

[43] Samuel Fingeret. 2015. *Defeating Code Reuse Attacks with Minimal Tagged Architecture*. Ph. D. Dissertation. MIT CSAIL.

[44] James C Foster, Vitaly Osipov, Nish Bhalla, Niels Heinen, and D Aitel. 2005. Buffer overflow attacks. *Syngress, Rockland, USA* (2005).

[45] Ronald Gil, Hamed Okhravi, and Howard Shrobe. 2018. There's a hole in the bottom of the C: On the effectiveness of allocation protection. In *Proceedings of the IEEE Conference on Secure Development (SecDev)*.

[46] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *IEEE S&P*.

[47] Julián Armando González. 2015. *Taxi: Defeating Code Reuse Attacks with Tagged Memory*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[48] J. Hiser, A. Nguyen, M. Co, M. Hall, and J.W. Davidson. 2012. ILR: Where'd My Gadgets Go. In *IEEE Symposium on Security and Privacy*.

[49] intel. 2013. Introduction to Intel Memory Protection Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions?language=es

[50] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. Hextype: Efficient detection of type confusion errors for c++. In *CCS '17*.

[51] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N.M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 641–648.

[52] Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2009. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*.

[53] Channoh Kim, Jaehyeok Kim, Sungmin Kim, Dooyoung Kim, Namho Kim, Gitae Na, Young H. Oh, Hyeon Gyu Cho, and Jae W. Lee. 2017. Typed Architectures: Architectural Support for Lightweight Scripting. In *Proceedings of the Twenty-Second International Conference on Architectural Support for*

Programming Languages and Operating Systems (ASPLOS). ACM Press, 77–90.

[54] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

[55] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[56] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-Pointer Integrity. (2014).

[57] Albert Kwon, Udit Dhawan, Jonathan Smith, Thomas Knight, and Andre Dehon. 2013. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 721–732.

[58] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*.

[59] Pavel Laskov et al. 2014. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 197–211.

[60] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert – a formally verified optimizing compiler. In *Embedded Real Time Software and Systems (ERTS)*.

[61] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices* 35, 11 (2000), 168–177.

[62] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[63] Tong Liu, Gang Shi, Liwei Chen, Fei Zhang, Yaxuan Yang, and Jihu Zhang. 2018. TMDFI: Tagged Memory Assisted for Fine-Grained Data-Flow Integrity Towards Embedded Systems Against Software Exploitation. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, New York, NY, USA, 545–550.

[64] lowRISC Team. 2017. Tag support in the Rocket core. https://www.lowrisc.org/docs/minion-v0.4/tag_core/

[65] A Theodore Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon W Moore, and Robert NM Watson. 2019. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals.. In *NDSS*.

[66] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing Kernel Hacks with HAKC. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, Vol. 22. 1–17.

[67] Arjun Menon, Subadra Murugan, Chester Rebeiro, Neel Gala, and Kamakoti Veezhinathan. 2017. Shakti-t: A risc-v processor with light weight security extensions. In *HASP '17*.

[68] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, Vol. 22. 1–17.

[69] David A Moon. 1985. Architecture of the Symbolics 3600. *ACM SIGARCH Computer Architecture News* 13, 3 (1985), 76–83.

[70] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2014. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 175.

[71] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices* 44, 6 (2009), 245–258.

[72] Hamed Okhravi. 2021. A Cybersecurity Moonshot. *IEEE Security & Privacy* 19, 3 (2021), 8–16. https://doi.org/10.1109/MSEC.2021.3059438

[73] Aleph One. 1996. Smashing the stack for fun and profit. *Phrack magazine* 7, 49 (1996), 14–16.

[74] OpenBSD. 2003. OpenBSD 3.3. http://www.openbsd.org/33.html

[75] Oracle. 2020. Sparc M8 Software in Silicon Features. https://docs.oracle.com/cd/E55211_01/html/E55216/gpzpn.html
[76] Elliott I Organick. 1973. *Computer system organization: the B5700/B6700 series*. Academic Press.
[77] Christian W Otterstad. 2015. A brief evaluation of Intel® MPX. In *Systems Conference (SysCon), 2015 9th Annual IEEE International*. IEEE, 1–7.
[78] Meltem Ozsoy, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Tameesh Suri. 2011. SIFT: A low-overhead dynamic information flow tracking architecture for SMT processors. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*. ACM, 37.
[79] Linux Manual Page. 2019. http://man7.org/linux/man-pages/man7/pkeys.7.html
[80] Christian Palmiero, Giuseppe Di Guglielmo, Luciano Lavagno, and Luca P. Carloni. 2018. Design and Implementation of a Dynamic Information Flow Tracking Architecture to Secure a RISC-V Core for IoT Applications. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*.
[81] V. Pappas, M. Polychronakis, and A.D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *IEEE Symposium on Security and Privacy*.
[82] Joseph A Paradiso and Thad Starner. 2005. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive computing* 4, 1 (2005), 18–27.
[83] PaX. 2003. PaX Address Space Layout Randomization. ttp://pax.grsecurity.net/docs/aslr.txt
[84] Stephen Phillips. 2014. M7: Next generation SPARC. In *Hot Chips 26 Symposium (HCS), 2014 IEEE*. IEEE, 1–27.
[85] Christian Pilato, Kaijie Wu, Siddharth Garg, Ramesh Karri, and Francesco Regazzoni. 2018. TaintHLS: High-level synthesis for dynamic information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2018), 798–808.
[86] Krerk Piromsopa and Richard J Enbody. 2006. Secure bit: Transparent, hardware buffer-overflow protection. *IEEE Transactions on Dependable and Secure Computing* 3, 4 (2006), 365–376.
[87] Joël Porquet and Simha Sethumadhavan. 2013. WHISK: An uncore architecture for dynamic information flow tracking in heterogeneous embedded SoCs. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 4.
[88] Raj Prakash. 2015. The Holy Grail - Real Time Memory Access Checking.
[89] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *Proceedings of IEEE Annual Computer Security Applications Conference (ACSAC'21)*.
[90] Nick Roessler and Andre DeHon. 2018. Protecting the Stack with Metadata Policies and Tagged Hardware. In *2018 IEEE Symposium on Security and Privacy*. IEEE, 478–495.
[91] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. 2017. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity.. In *NDSS*.
[92] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. Address-Sanitizer: A Fast Address Sanity Checker.. In *USENIX Annual Technical Conference*. 309–318.
[93] Ryota Shioya, KIM Daewung, Kazuo Horio, Masahiro Goshima, and Shuichi Sakai. 2011. Low-overhead architecture for security tag. *IEICE transactions on information and systems* 94, 1 (2011), 69–78.
[94] Howard Shrobe, Andre DeHon, and Thomas Knight. 2009. *Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (TIARA)*. Technical Report. DTIC Document.
[95] Howard Shrobe, Thomas Knight, and Andre de Hon. 2007. TIARA: Trust Management, Intrusion-tolerance, Accountability, and Reconstitution Architecture.
[96] Asia Slowinska and Herbert Bos. 2009. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European conference on Computer systems*.
[97] Burton J Smith. 1982. Architecture and applications of the HEP multiprocessor computer system. In *25th Annual Technical Symposium*. International Society for Optics and Photonics, 241–248.
[98] Charles Smutz and Angelos Stavrou. 2012. Malicious PDF detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 239–248.
[99] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-Assisted Data-flow Isolation. In *IEEE Symposium on Security and Privacy*.
[100] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: sanitizing for security. In *Proc. of IEEE Symposium on Security and Privacy*.

[101] Wei Song. 2015. lowRISC tagged memory tutorial. https://www.lowrisc.org/docs/tagged-memory-v0.1/

[102] Chad Spensky, Aravind Machiry, Nathan Burow, Hamed Okhravi, Rick Housley, Zhongshu Gu, Hani Jamjoom, Christopher Kruegel, and Giovanni Vigna. 2021. Glitching Demystified: Analyzing Control-flow-based Glitching Attacks and Defenses. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 400–412. https://doi.org/10.1109/DSN48987.2021.00051

[103] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. 2009. Breaking the memory secrecy assumption. In *Proc. of EuroSec'09*. 1–8.

[104] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*. ACM, 160–171.

[105] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. In *Acm Sigplan Notices*.

[106] Gregory T Sullivan, André DeHon, Steven Milburn, Eli Boling, Marco Ciaffi, Jothy Rosenberg, and Andrew Sutherland. 2017. The Dover inherently secure processor. In *Technologies for Homeland Security (HST), 2017 IEEE International Symposium on*. IEEE, 1–5.

[107] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proc. of IEEE Symposium on Security and Privacy*.

[108] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* (1984).

[109] Mohit Tiwari, Xun Li, Hassan MG Wassel, Frederic T Chong, and Timothy Sherwood. 2009. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *MICRO '09*.

[110] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *ASPLOS '09*.

[111] Neil Vachharajani, Matthew J Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A Blome, George A Reis, Manish Vachharajani, and David I August. 2004. RIFLE: An architectural framework for user-centric information-flow security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*. IEEE, 243–254.

[112] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1675–1689.

[113] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2008. Flexitaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 173–184.

[114] John Von Neumann. 1993. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing* (1993).

[115] Bryan Ward, Richard Skowyra, Chad Spensky, Jason Martin, and Hamed Okhravi. 2019. The Leakage-Resilience Dilemma. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*.

[116] Robert NM Watson, Robert M Norton, Jonathan Woodruff, Simon W Moore, Peter G Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, et al. 2016. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro* 36, 5 (2016), 38–49.

[117] Robert NM Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, Munraj Vadera, and Khilan Gudka. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*.

[118] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V.. In *NDSS*.

[119] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. *Mondrian memory protection*. Vol. 30. ACM.

[120] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 2005. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *ACM SIGOPS Operating Systems Review*. ACM.

[121] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International*

Symposium on Computer Architecuture (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 457–468. http://dl.acm.org/citation.cfm?id=2665671.2665740

[122] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. 2005. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the 12th ACM conference on Computer and communications security*. 223–234.

[123] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2008. Hardware Enforcement of Application Security Policies Using Tagged Memory.. In *OSDI*, Vol. 8. 225–240.

[124] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 559–573.

[125] Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Larus. 1987. *SPUR Lisp: Design and implementation*. Technical Report. Technical Report UCB/CSD 87/373, Computer Science Division (EECS), University of California, Berkeley.