CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities

Rasool Sharifi
University of Virginia
as3mx@virginia.edu

Ashish Venkat
University of Virginia
venkat@virginia.edu

***Abstract*—This work introduces the CHEx86 processor architecture for securing applications, including legacy binaries, against a wide array of security exploits that target temporal and spatial memory safety vulnerabilities such as *out-of-bounds* accesses, *use-after-free*, *double-free*, and *uninitialized reads*, by instrumenting the code at the microcode-level, completely under-the-hood, with only limited access to source-level symbol information. In addition, this work presents a novel scheme for speculatively tracking pointer arithmetic and pointer movement, including the detection of pointer aliases in memory, at the machine code-level using a configurable set of automatically constructed rules. This architecture outperforms the address sanitizer, a state-of-the-art software-based mitigation by 59%, while eliminating porting, deployment, and verification costs that are invariably associated with recompilation.**

***Index Terms*—Memory Safety, Capabilities, Microcode**

I. Introduction

The modern computing landscape has witnessed the rapid growth, expansion, and deployment of increasingly complex software, amid the relentless pursuit of high performance. This rise in software complexity has shown to proliferate vulnerabilities at an alarming rate, opening the door to a myriad of high-impact security exploits [25], [67], [82]. Notably, software flaws that arise due to heap memory safety violations such as *out-of-bounds* accesses, *use-after-free*, and *uninitialized reads* have been principal anchor points for a number of security exploits in-the-wild [9], [30], [43], [47], [63]. In fact, memory safety violations have consistently accounted for about 70% of the vulnerabilities patched via security updates every year, as reported in recent studies by Microsoft and Google [30], [47] (re-created in Figure 1), highlighting the need for stronger and more robust exploit mitigations that can effectively combat memory safety violations, while maintaining high performance and high programmability.

The literature abounds with *capability-based addressing* schemes that have shown to effectively curtail many temporal and spatial memory safety exploits [1], [10], [14], [23], [24], [29], [34], [40], [50], [81], [86], by requiring all memory accesses to occur via capabilities or fat-pointers that contain *unforgeable* bounds and permissions information alongside the actual address, thereby restricting traditional C-style pointers to operate within their respective allocated regions and forbidding stray accesses that lack sufficient privileges. These systems abide by a core tenet of computer security – the *principle of least privilege*, that stipulates that every user and program operate with the least set of privileges required to perform a task. RISC architectures have shown to be particularly viable for implementing such a scheme, given that only a handful of RISC instructions directly operate on memory, thereby considerably limiting the number of intrusive changes

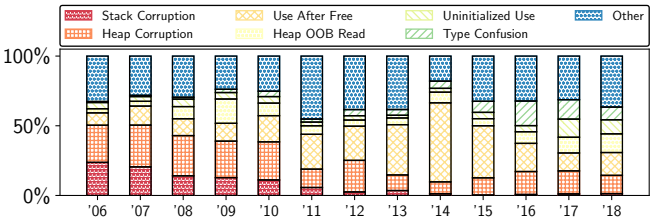


Fig. 1: Root Cause of CVEs by Patch Year (since 2006). The "other" category consists of XSS/zone elevation, DLL planting, and file canonicalization and symbolic link issues.

to both the ISA and the processor pipeline, as demonstrated recently by the CHERI architecture [14], [23], [34], [81], [86].

However, deploying such capability-based protection schemes on CISC architectures requires non-trivial modifications to the ISA, notwithstanding extensive porting, deployment, and verification costs that come with it, leaving large corpora of binaries unprotected. In this paper, we explore a suite of novel, high-performance, and transparent capability-based protection mechanisms, dubbed CHEx86 (Capability Hardware-Enhanced x86), to secure unmodified source and object code, completely *under-the-hood*, against a wide array of temporal and spatial memory safety exploits, without the need for software patching and recompilation.

This paper shows that such a transparent capability-based protection scheme can be seamlessly enforced in a variety of different design configurations – (a) *hardware-only*, (b) *binary translation-driven*, and (c) *microcode-level instrumentation*, and that they each offer different design and performance trade-offs. The hardware-only scheme completely forgoes dynamic code instrumentation, and instead delegates the load/store unit to perform capability checks before initiating the memory access. On the other hand, the binary translation-driven scheme does not require any modifications to the load/store unit, but dynamically instruments every x86 instruction that employs a register-memory addressing mode, with either software-based capability checks or special instructions made available through secure ISA extensions.

In stark contrast to both these schemes, the microcode-level capability enforcement offers a unique design point – it takes advantage of an already existing layer of indirection implemented in modern ISAs – the CISC-to-RISC micro-op translation interface – to dynamically instrument the micro-op stream with capability checks on-demand. There are substantial benefits to such an undercover injection of capability checks at the microcode level. First, it opens the door to selective and context-sensitive enforcement [72]–[75] of capability-based protection, allowing the micro-op stream to be in-

strumented on-demand, only while executing security-critical code, providing significant flexibility over existing hardware-only protection mechanisms that can either be always-on or always-off. Second, it allows capability checks to be customized as necessary by leveraging existing mechanisms in modern processors to re-route machine instruction translation to the microcode RAM [38], [73], further enabling hardware vendors to deploy unobtrusive field updates in response to zero-day attacks, without the need for extensive software patching. Third, a microcode-level injection mechanism not only addresses memory safety vulnerabilities that are inherent to the software code in execution, but interestingly, also severely inhibits transient execution attacks, such as Spectre [37], that hinge on bypassing software-based bounds checks by exploiting speculative execution.

To make this vision of a transparent capability-based addressing scheme a reality, we explore two major extensions to conventional processor architectures. First, to provide efficient capability-based protection under-the-hood, CHEx86 designs store and manage memory capabilities in distinct shadow tables, requiring privileged access. In the microcode variant of CHEx86, access is restricted to micro-ops that are dynamically generated (outside of existing translation) by the microcode engine. This is similar in spirit to many traditional capability systems and hardware-only pointer safety mechanisms that place capabilities and data in separate memory regions [1], [24], [36], [50], [84]. Second, we introduce *speculative pointer tracking*, a novel architectural technique, implemented entirely within the processor's front-end, that allows us to trap allocation and de-allocation events, and further dynamically track pointer arithmetic, pointer movement, and spilled pointer aliases on-the-fly, using an automatically generated and configurable low-level *peephole rule* database, coupled with an address prediction mechanism that tracks pointers being reloaded from memory. Since this approach does not require any dynamic code instrumentation to track pointers, we significantly accelerate pointer tracking, even for pointer-intensive applications.

In contrast to most existing capability-based protection mechanisms that rely on extensive pointer analyses and transformations at the compiler-level to track and maintain a pointer's allocated bounds, scope, and privileges, the speculative pointer tracking scheme we introduce, operates entirely within the processor and thereby lacks access to source-level information. While this may seemingly impede our ability to track certain code paths and leverage language-level semantics that are only available for analysis at compile-time, in this paper, we show that even a small set of peephole rules can effectively address several high-impact vulnerabilities including *out-of-bounds* accesses, *use-after-free*, *double free*, and *invalid free* among others. Furthermore, to the best of our knowledge, this is the first work to establish the viability of employing hardware-based prediction mechanisms to track pointers, based on the hypothesis that the temporal pointer access patterns of many applications are highly predictable – most code regions typically, and often repeatedly, access only a limited set of pointers, in a predictable sequence.

We note that such a transparent pointer tracking scheme places us at a significant advantage over prior approaches because it now allows us to secure several previously unprotected legacy binaries, while significantly minimizing the performance impact due to software-only pointer tracking, and simultaneously eliminating a substantial amount of development, verification, and deployment costs that are invariably associated with recompilation and/or porting of legacy code. It also, more importantly, lays the groundwork for performing other program analyses and transformations in hardware, such as low-level type verification and data-oblivious code transformations [21], [48], [88].

This work makes the following major contributions.

- We extend conventional CISC architectures to provide transparent capability-based protections, safeguarding vulnerable applications (including legacy binaries), against a wide range of security exploits that target temporal and spatial memory safety, without compromising software compatibility.
- We identify several design points for implementing capability-based addressing, at the hardware-level, binary translation-level, and the microcode-level, and evaluate their performance and design trade-offs.
- We introduce a novel speculative pointer tracking scheme that allows us to track pointer arithmetic, pointer movement, and spilled pointer aliases, completely under-the-hood, with access to limited source-level information.
- We show that most code regions can be characterized by temporal pointer access patterns that are remarkably predictable, and further demonstrate that a simple hardware-based prediction scheme can accurately identify pointer operations at run-time.
- Owing to our flexible micro-op instrumentation and speculative pointer tracking scheme, we provide significantly high performance in comparison to existing software-based defenses, and in particular we outperform software-only schemes by 59% (SPEC) to 2.2X (PARSEC), while incurring only 9% (PARSEC) to 14% (SPEC) degradation with respect to an insecure baseline.

II. Background and Related Work

Memory Safety. The C and C++ languages have been widely adopted in systems programming, due to their high flexibility and performance-oriented nature, an undesirable side effect of which, is that stray accesses by C/C++ pointers into privileged memory simply go unchecked, due to the lack of language-level enforcement of memory safety. Depending upon the type of illegitimate pointer access, memory safety violations can be broadly classified as: (a) *spatial safety violations* that occur when a pointer dereference results in a memory access outside of its allocated bounds, and (b) *temporal safety violations* that occur as a result of dereferencing a pointer that points to invalid (typically, unallocated or freed) memory. While most memory safety vulnerabilities typically manifest as a direct consequence of unsafe programming practices, a significant chunk of temporal safety vulnerabilities also arise due to exploitable implementations of high-performance heap management libraries [6], [9], [26], [53], [67].

Tripwire-Based Mitigations. Tripwires are a class of memory safety mitigations that prevent memory trespassing by placing *redzones* between allocated blocks of memory, which then get activated upon an illegitimate pointer access. Notable

commercial implementations of tripwire-based approaches include Google's AddressSanitizer (ASan) [65] and Valgrind's memory checker [52]. The key distinguishing aspect of tripwires is that they get immediately activated upon a memory trespass, unlike canaries [18], [19], [27] that need to be explicitly checked by software. Despite providing strong security guarantees, tripwire-based approaches incur prohibitively high performance overheads, rendering them unfavorable for deployment in performance-conscious installations. More recent work on hardware-assisted tripwire mechanisms such as SafeMem [61], REST [68], and Califorms [63], however, have made significant strides at effectively hiding the run-time overheads incurred due to static instrumentation.

Fat Pointers. Several prior efforts have proposed safe language extensions to C/C++ via *fat pointers* that maintain the object's metadata (typically, bounds and permissions information) alongside the address itself, enabling both static verification and run-time dynamic checks to flag memory accesses that are out-of-context. Some early and noteworthy mentions include Cyclone [33] and CCured [51]. However, inlining of the metadata along with the address results in loss of backwards compatibility. In contrast, approaches like Hardbound [24], Softbound [50], Watchdog [49], Intel's MPX [54], and BOGO [92] maintain metadata in disjoint shadow memory that is looked up for every dereferencing operation to perform dynamic checks in software or in hardware using micro-ops. However, these shadow structures tend to substantially grow in size for pointer-intensive applications, resulting in high metadata lookup overheads. Baggy Bounds [1], [44] considerably mitigates this overhead by leveraging the Jones and Kelly approach [35] to metadata storage.

While we bear some similarities to some of these approaches, we also differ in several key aspects. First, we provide both spatial and temporal safety, unlike Hardbound, Softbound, Baggy Bounds, and Intel's MPX that only address spatial errors. Second, these mechanisms rely on tracking pointer arithmetic via dynamic code instrumentation at the software or micro-op level, resulting in significant slowdown for pointer-intensive applications. In contrast, we present novel mechanisms to speculatively track pointers under-the-hood with full alias detection, using minimal source-level symbol information, without run-time instrumentation of code. Finally, our approach is significantly more flexible due to the fact that it can enforce on-demand and context-sensitive capability protection for security-critical code, while most of the prior approaches are either always-on or always-off. We present a more detailed compare and contrast in Table IV.

Capability-Based Systems. The literature describes several capability-based operating systems and hardware architectures. These systems are guided by a common philosophical doctrine – the *principle of least privilege* – that stipulates every user and program to operate with the least set of privileges required to perform a task. This principle is primarily enforced via *capabilities*, that are essentially pointers to a given system resource that provide appropriate access rights to their owners. An important trait of these capabilities is that they are *unforgeable*, which means they can be created, transferred, and destroyed, but cannot be modified. Notable modern OS-level capability-based implementations include the seL4 microkernel [36] and the Barrelfish multikernel [4].

Modern capability-based hardware architectures such as the M-Machine [10], low-fat pointers [40], and CHERI [14], [23], [34], [81], [86], rely on *tagged memory* to implement *unforgeable fat-pointers*, allowing for the metadata (bounds and permissions) to be maintained inline with the address, thereby eliminating expensive shadow table lookups, by trading off area, storage, and memory bandwidth. In particular, the CHERI capability model has made major strides in establishing the viability of implementing practical capability-based addressing in RISC-style architectures, operating systems, and runtime systems. However, it necessarily requires applications to be ported and recompiled to the CHERI architecture, leaving a significant chunk of legacy applications and traditionally CISC execution environments unprotected.

Low-Level Secure Code Instrumentation. Several proposals in the past have shown that low-level secure code instrumentation is particularly effective at enforcing security policies such as control-flow integrity and memory safety. DISE [15]–[17] provides a framework for instrumenting the dynamic instruction stream in hardware, at the microcode-level, to perform simple bounds checking and shadow stack protection. More recently, there has been a renewed interest in on-demand microcode-level customization [38], [72]–[75] of machine code to defend against side-channel and transient execution attacks.

Hardware-Based Information Flow Tracking. The speculative pointer tracker described in this work is built on top of a extensive body of prior work on information flow tracking. Suh, et al [69] first described and proposed architectural mechanisms for Dynamic Information Flow Tracking (DIFT) to tag data from untrusted channels such as the console, network, and filesystem, as potentially spurious, and then tag the result of a computation (registers and memory locations) as spurious if it was computed using one or more spurious inputs, by following the dynamic information flow in instructions, and thereby restricting information flow of such spurious values with a suitable pre-defined security policy. Information flow tracking techniques have been described at a variety of different levels in hardware – from the gate-level to the microarchitecture-level, offering varying degrees of trade-offs [11]–[13], [20], [22], [76], [77], [80], [91]. More recently, Yu, et al [89] introduced Speculative Taint Tracking to analyze taint/secret propagation in the context of transient execution attacks.

Low-Level Program Analysis. Morrisett, et al [48] demonstrate the feasibility of translating programs written in a high-level functional programming language to a type-preserving assembly language, based on a simple RISC-style instruction set, fully automating the process of generating proof-carrying code to enforce security properties at the machine code-level. Crary, et al. [21] extend this type-preserving translation scheme to emit x86 assembly instructions annotated with type information. Further, Azevedo de Amorim, et al. [3] describe a system that is able to track type information in hardware, by leveraging tagged memory. More recently, Mchmahan, et al. [45] present Bouncer, a hardware engine for performing static analysis in a microcontroller architecture implementing the ZARF functional ISA [46]. While the pointer-tracking scheme we describe is similar in spirit to classic Dynamic

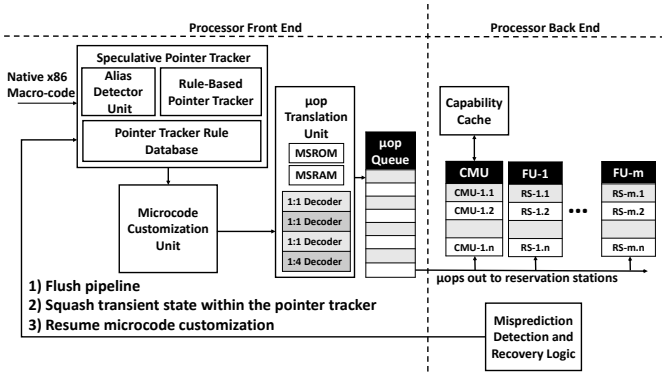


Fig. 2: Architectural Overview of CHEx86

Information Flow Tracking (DIFT) [69], we also extend these systems, in many novel ways, to more efficiently deal with speculative execution and aliases.

III. Assumptions and Threat Model

Threat Landscape. The goal of this work is to secure unmodified binaries against exploits that target *temporal* and *spatial* memory safety violations, such as *out-of-bounds* accesses, *use-after-free*, *double free*, *invalid free*, and other *metadata corruption* errors in the heap and the global data section. The granularity of protection we provide is at the object-level, and our threat model does not yet include attacks that exploit intra-object spatial errors (e.g., overflowing into an adjacent field within a struct). Furthermore, we do not assume the presence of any more defenses in our execution environment, than what is already available in conventional hardware architecture and operating system implementations (e.g., ASLR [57] and DEP [58]).

Attacker Characterization. We assume that, in absence of our defense mechanism, the attacker has the ability to repeatedly exploit one of more memory safety vulnerabilities in the program code, in order to gain arbitrary read, write, and execute privileges. Further, we assume that the attacker has complete access to the source code of the program in execution, along with any other compile-time metadata that may assist in the automatic generation of exploits [2], [31], [62], [64], [90]. However, like other contemporary approaches [63], we assume that the executable binary itself is protected (e.g., via code-signing) and cannot be tampered with.

In addition, we assume that the attacker neither has access to the dynamically injected capability check micro-ops (since they reside outside any user-addressable memory), nor to the shadow tables that maintain capability and alias metadata (since they may only be accessed by dynamically generated micro-ops injected by the microcode engine).

Trusted Components. We assume that all hardware components including the ones we add in this work are trusted and tamper-resistant [70], [71]. We also include any privileged entities that have the ability to perform microcode updates and/or configure parameters of the underlying microcode-level defense framework, in our TCB. Furthermore, our threat model excludes attacks [39] that can reverse engineer or poison the microcode as these attacks can only be carried out on old AMD processors (K8 and K10) that don't have advanced microcode update signing protections in place.

Side-Channel Attacks. Finally, since our approach does not provide any explicit protections against side-channel and transient execution attacks, we exclude them from our threat model. Interestingly, our mechanisms do protect against the Spectre-v1 attack [37] that attempts to bypass software-based bounds checks by exploiting side effects of speculative execution. This is due to the fact that our capability checks are essentially part of the same macro-op that performs the dereferencing operation, and therefore, in most cases cannot be bypassed. However, we note that this depends upon the TOC/TOU assumptions of the underlying implementation.

IV. The CHEx86 Architecture

A. Architectural Overview

Goals and Challenges. One of the main goals of CHEx86 is to provide high performance and transparent capability-based protections to secure unmodified binaries against a wide array of temporal and spatial memory safety exploits. However, this entails solving several key challenges.

First, to instrument the dynamic instruction stream with appropriate capability checks, the processor's front-end should be able to seamlessly and transparently track pointer arithmetic, pointer movement, and pointer dereferencing operations. On RISC architectures that expose a limited number of opcodes and addressing modes, this may be accomplished by using a small set of pointer-tracking rules. However, the nature, number, and type of such rules required to track pointers on CISC architectures like x86 is immediately unclear – a major contribution of this research is a principled approach to constructing such a pointer-tracking rule database for x86.

Second, in addition to tracking pointer movement between registers, the front-end should also be able to track spilled pointers in memory and detect instances of such spilled pointers getting reloaded into registers. These typically arise when a register containing a pointer is spilled to the stack, the global data section, or the heap memory, essentially creating a spilled pointer alias in memory. Note that this is very common in code regions that use pointers with multiple levels of indirection. Prior research has dealt with this problem in two primary ways – (a) by inlining of capability metadata for both in-register and in-memory pointer aliases (e.g., CHERI), which necessarily requires recompilation, and (b) by associating every word in memory with pointer identifier metadata in shadow memory and instrumenting every load to also read the associated pointer identifier metadata from shadow memory (e.g., Watchdog), not only deferring pointer alias detection until the execute stage, but incurring significant overheads in performance and storage. A key contribution of this work is a *speculative pointer tracker* that employs a simple prediction scheme to detect spilled pointer aliases at the front-end.

Third, to emulate fat pointers at the microcode level, completely under-the-hood, it is important that the processor maintains capability metadata in shadow structures, while making minimally intrusive changes to the core design of the processor pipeline, and especially while minimizing the impact on the critical path of load operations. This work employs several microarchitectural optimizations (including the use of shadow caches) to minimize the impact of such shadow metadata structures on performance.

Finally, the pointer tracker employed by CHEx86 is inherently speculative in nature as (1) it employs a predictor to detect spilled pointer aliases, and (2) all pointer tracking is done at the front-end of the pipeline on speculatively fetched instructions. Therefore, it is important to ensure that the pointer tracker gracefully recovers from any form of misspeculation, including branch mispredictions and spilled alias mispredictions. While we leverage conventional squashing mechanisms to recover from mispredictions, we describe a few additional optimizations in Sections V-C and V-D.

Our Approach. Figure 2 provides the architectural overview of CHEx86. The principal components include: (a) the microcode customization unit (applicable only for the microcode variant of CHEx86), (b) the shadow capabilities table, and (c) the speculative pointer tracker. At a high level, the speculative pointer tracker is responsible for tracking pointers, including spilled pointer aliases, at the front-end of the pipeline. It acts as a trigger mechanism to the microcode customization unit that instruments pointer dereferencing operations with appropriate capability checks by looking up the shadow capabilities table.

B. Principal Components of CHEx86

Microcode Customization. The microcode customization unit implements the core functionality of the microcode variant of CHEx86 by instrumenting the dynamic instruction stream with appropriate capability generation, validation, and free micro-ops. This can be done by leveraging existing functionality and existing micro-op instructions in modern processor architectures (including ARM and x86), by simply re-routing the translation of relevant native macro-operations to the microcode RAM that hosts custom translations [38], [73]–[75].

Shadow Capabilities Management. The shadow capability table is a per-process table that stores the list of all capabilities granted to the program in execution, essentially tracking all live and freed memory allocations (both static and dynamic). This table is located in a separate shadow address space that is only accessible by privileged code. Each entry of a shadow capability table contains a capability that is tagged by a non-zero unique capability identifier (denoted in this work, as PID).

In our implementation, each capability is 128 bit-wide, with 64 bits allocated to the *base address* and 32 bits allocated to *bounds*. The remaining 32 bits are used to maintain *permissions*, including *read*, *write*, *execute*, *busy*, and *valid* bits. While the *busy* bit indicates whether the memory block pointed to by the capability is currently in process of being allocated/freed, the *valid* bit indicates whether the capability points to a valid block of memory, typically used to detect *use-after-free* scenarios.

Since the shadow capability table maintains all capabilities, allocated and freed, throughout the program's execution, looking up the table at run-time for performing checks can be particularly expensive, especially given that every pointer dereferencing operation is predicated upon the result of a capability check. However, we find that the capability table lookup can be significantly accelerated by caching only those capabilities that are currently-in-use, with the help of a small in-processor capability cache. This is motivated by the fact that programs typically tend to frequently use only a handful of pointers at any given point of time, despite making several allocations over the course of their execution.

Figure 3 shows this phenomenon using three key metrics: (1) the total number of allocations made by an application over the course of its execution, (2) the maximum number of live (allocated, bur not freed) allocations at any given point of time, and (3) the average number of allocations that are actually in use during any given 100 million instruction interval. These statistics were collected by profiling C and C++ applications from the SPEC CPU2017 and PARSEC 2.1 benchmark suites, using *valgrind* [52]. We note that although an application may potentially make tens of millions of allocations over its lifetime, the maximum number of live allocations is an order of magnitude lower, and the number of allocations in use at any given point of time is actually much lower, by at least three orders of magnitude. In the average case, the number of allocations in use during a 100 million dynamic instruction interval (which is a considerably long interval, tens of milliseconds if not more) is only 7034, clearly motivating the utility of a small in-processor capability cache that can track allocations in use. To this end, we include a 64-entry fully associative capability cache in our CHEx86 design.

Note that the capability cache access is not on the critical path of load operations. In fact, in the microcode variant of CHEx86 (which is our default implementation), the capability cache is accessed only by special capability check micro-ops as described in detail in the next section.

Speculative Pointer Tracking. The speculative pointer tracker is responsible for tracking the transfer of capabilities between registers due to pointer arithmetic and pointer movement, in the front-end, while they are getting streamed in to the microcode engine. In addition, it also detects spilled pointer aliases with the help of a simple prediction mechanism, and identifies relevant memory dereferencing operations that need instrumentation. Section V describes the design of our speculative pointer tracker in more detail.

C. Functional Overview

In this section, we discuss how CHEx86 intercepts and handles memory allocation events, pointer arithmetic, pointer dereferencing, and pointer free events to *generate*, *transfer*, *validate*, and *free* capabilities respectively, by elucidating certain key events of a program in execution. We specifically take the example of our microcode variant to describe our instrumentation mechanisms, but note that this instrumentation may also happen with the help of a binary translator [78], [79]. In our hardware-only variant, we forgo instrumentation, but we perform our checks as part of the regular load/store micro-op.

Initial Configuration. At the time of scheduling a process on a CHEx86 core, the OS kernel or other trusted entities may configure a set of model-specific registers (MSRs) to *register* the instruction address of the entry and exit points of key heap management functions (e.g., *malloc*, *calloc*, *free*, *realloc*, etc.) that the program is expected is use, along with their respective signatures (recorded as a vector of architectural register names). On ASLR-enabled systems, the randomized addresses of the entry/exit points will be configured in the MSRs by the OS kernel, allowing us to intercept allocation/free events to appropriately generate/free capabilities. We intercept both entry and exit points because we want to keep track of both

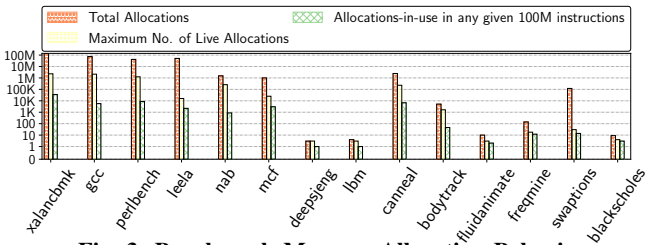


Fig. 3: Benchmark Memory Allocation Behavior

instantiation and completion of the allocation/de-allocation events. Note that these MSRs are saved and restored upon a context switch. As a result, there is a model-specific limit on the number of entry/exit points that can be registered per process.

Furthermore, at the time of process creation and program loading, the OS kernel may also load the *symbol table* into memory, if available, and further instruct CHEx86 (again, using a privileged *wrmsr* instruction) to initialize the shadow capability table by generating a capability for each global data object found in the symbol table, while appropriately recording their bounds and permissions information. Although, in this work, we only initialize our shadow tables using the information available from symbol tables, we note that our approach is flexible enough to be configured with metadata derived from more sophisticated static analysis.

Generation of capabilities. When a program in execution performs a memory allocation request by calling a *registered* heap management function such as *malloc*, the microcode customization unit immediately intercepts such an event, and further instruments the intercepted macro-op to include a special *capGen.Begin* micro-op that instantiates a new capability with the *busy* bit set, indicating that the allocation is still in progress. The micro-op also copies the contents of the argument register *%rdi* into the *bounds* field of the capability. Note that these micro-ops operate within the realms of speculative execution, and therefore updates to a transient capability register are only finalized upon commit.

Once the memory allocation is complete and the program is ready to return from the heap management function, the microcode customization unit again intercepts the event, since it is a registered exit point. This time, the intercepted macro-op is instrumented with a *capGen.End* micro-op that signals the completion of the memory allocation by resetting the *busy* bit. The micro-op also copies the contents of the return value register *%rax* to the *base address* field of the capability. Further, the micro-op also sets the *valid* bit of the capability, indicating the legitimacy of the allocation, iff the *base address* has been set to a non-zero value. At this point, the capability generation is complete, allowing no further modifications to the generated capability. Note that this two-step update procedure is only required by the microcode variant of CHEx86, which performs the instrumentation under-the-hood, without recompilation or binary translation. If the instrumentation is performed by a binary translator, it is possible to leverage special capability generation instructions exposed via ISA extensions, to perform this update in one step.

Transfer of capabilities. Upon returning from the memory allocation function, if the program executes an instruction to move the address contained in the result register *%rax* to a different register or is spilled to a memory location, a capability transfer should occur. In CHEx86, this capability transfer occurs implicitly via the speculative pointer tracker, which maintains its own set of tables to associate a register or a spilled memory address with a PID. Section V describes our speculative pointer tracker in detail.

Validation of capabilities. When the speculative pointer tracker detects a dereferencing operation carried out using base register that is tagged with a non-zero PID, the microcode customization unit instruments the dereferencing operation to include a special *capCheck* micro-op, which during execution, looks up the shadow capability table using the appropriate PID, and further checks for bounds/permission violations.

Freeing of capabilities. Similar to capability generation, freeing of capabilities also happens in two steps. Upon interception of the entry point of a registered de-allocation function, a *capFree.Begin* micro-op is injected. With the help of the speculative pointer tracker, the PID of the argument register *%rdi* is obtained, and is used by the *capFree.Begin* micro-op to look up the shadow capability table and set the *busy* bit, indicating that a de-allocation is in progress. Once it is time to return from the de-allocation function, an exit point is intercepted, resulting in the injection of a *capFree.End* micro-op that resets both the *valid* and the *busy* bit, signalling the freeing of the capability. We continue to maintain the capability in our shadow capability table and track transfers via the speculative pointer tracker, to detect *use-after-free* violations.

In multithreaded environments, when a pointer is freed on one core, invalidate requests are sent to all other cores in the system, to ensure that the *valid* and the *busy* bit of the capability entries corresponding to that pointer are reset across all in-processor capability caches in the system. Note that, due to the unforgeability property of capabilities, these invalidation requests have to be sent only once at the time of freeing capabilities.

V. Speculative Pointer Tracking

This section describes the key mechanisms and functionalities of our speculative pointer tracker, including rule-based pointer tracking, alias detection, and misspeculation recovery.

A. Rule-Based Pointer Tracking

Tracking pointer activity in machine code, in the absence of any type annotations, is a well-known difficult problem. This is due to the loss of precise language-level semantics during the translation to low-level machine code. In this work, we show that capability transfers due to pointer arithmetic, pointer movement, and pointer dereferencing operations can be efficiently tracked at the decoder-level using a small configurable set of rules.

The goal of our rule-based pointer tracker is to propagate capabilities from the source operands of a micro-op instruction to its destination operand, based on the instruction's opcode, addressing mode, and the current set of capabilities associated with its operands. However, given the multitude of different ways in which a pointer can be manipulated, characterizing pointer activity using a limited set of low-level rules can be a non-trivial endeavor. While it is possible to construct the rule database (and update it, as necessary) with the help of an

TABLE I
POINTER TRACKING RULE DATABASE

μop	Addr. Mode	Example	Capability Propagation	Code Example
MOV	Reg-Reg	mov %rcx, %rbx	PID(rcx) ← PID(rbx)	ptr1 = ptr2;
AND	Reg-Reg	and %rcx, %rbx, %rax	if (PID(rax) == 0) then PID(rcx) ← PID(rbx) if (PID(rbx) == 0) then PID(rcx) ← PID(rax)	int mask = 0xffff0000; ptr2 = ptr1 & mask;
		If the PID of one source operand is zero, then assign the PID of the other source operand to the destination operand		
	Reg-Imm	andi %rcx, %rbx, $imm	PID(rcx) ← PID(rbx)	ptr2 = ptr1 & 0xffff0000;
LEA	Reg-Reg	lea %rcx, (%rbx, %idx, scl)	PID(rcx) ← PID(rbx)	ptr = &a[50];
ADD	Reg-Reg	add %rcx, %rbx, %rax	if (PID(rax) == 0) then PID(rcx) ← PID(rbx) if (PID(rbx) == 0) then PID(rcx) ← PID(rax)	ptr2 = ptr1 + const;
		If the PID of one source operand is zero, then assign the PID of the other source operand to the destination operand		
	Reg-Imm	addi %rcx, %rbx, $imm	PID(rcx) ← PID(rbx)	ptr2 = ptr1 + 4
SUB	Reg-Reg	sub %rcx, %rbx, %rax	PID(rcx) ← PID(rbx)	ptr2 = ptr1 - const;
		Always assign the PID of the second operand to the destination operand		
	Reg-Imm	subi %rcx, %rbx, $imm	PID(rcx) ← PID(rbx)	ptr2 = ptr1 - 4;
LD	Reg-Mem(qw)	ldq %rcx, [EA]	PID(rcx) ← PID(Mem[EA])	int **ptr1 = malloc(400*100); int *ptr2 = ptr1[100];
ST	Reg-Mem(qw)	stq %rcx, [EA]	PID(Mem[EA]) ← PID(rcx)	int **ptr1 = malloc(400); int *ptr2 = malloc(100); *ptr1 = ptr2
MOVI	Reg-Imm	limm %rax, $imm	PID(rax) ← PID(-1)	int *p = (int *)0x7fff1000;
All other operations			PID(result) ← PID(0)	

expert, in this work, we automate the process of incrementally constructing our rule database, with the help of a hardware checker co-processor that validates our rules at run-time. This process does not require any support from the compiler.

The rule database is first initialized to a small set of rules by an expert, and is then validated and incrementally updated in an offline profiling step with the help of the hardware checker. The operation of the hardware checker is two-fold. First, for every micro-op in execution, it exhaustively looks up our shadow tables to check if the result of the instruction is an address that points to any of the allocated/freed blocks that we have been tracking. If the search fails, it concludes that the result operand is either not a pointer, or that it points to a memory block that is not of interest to us (e.g., memory allocated using an *unregistered* heap management function, or a global structure that wasn't listed in the symbol table). If the search succeeds, then it records the PID of the corresponding memory block. The hardware checker validates our rules by checking the predicted PID from the speculative pointer tracker against the actual PID obtained via the exhaustive search, and if this check fails, it dumps the offending instruction along with its execution state, and requests that the rule database be updated via manual intervention.

Table I presents our rule database that was automatically constructed using the above process, while running C and C++ benchmarks from the SPEC and PARSEC suites [55], the RIPE security suite [83], LLVM's Address Sanitizer test suite [65], and the How2Heap suite of heap exploits from the CTF team ShellPhish. This shows that pointer activity can be successfully tracked in hardware using a limited of rules, confirming our hypothesis that most high-level pointer manipulation methods boil down to only few distinct micro-op execution patterns. But more importantly, we now have a framework to extend this pointer tracking rule database, as we execute new workloads.

Our approach does not yet explicitly deal with the problem of integer provenance, as most existing capability-based systems [85] defer to the compiler to detect and flag wild pointer dereferences performed using an integer constant virtual address. However, just to exemplify the extensibility of our rule database, consider the rule MOVI in Table I, which shows a load-immediate micro-op that moves an integer immediate value to a register. This rule assigns a special PID(-1) to the destination register, allowing us to track any indirect memory reference performed using that register. Further, the capabilities table will not include any entry that corresponds to PID(-1), as no such capability was generated using a registered allocation function, allowing *capCheck* micro-ops to flag such violations.

TABLE II
TEMPORAL POINTER ACCESS PATTERNS

Pattern	*Stride*	*Example PIDs*						
Constant	0	31	31	31	31	31	31	31
Stride	3	13	16	19	22	25	28	31
Batch + Stride	4	11	11	11	15	15	15	15
Batch + No Stride	NA	22	22	22	13	99	99	99
Repeat + Stride	1	26	27	28	26	27	28	26
Repeat + No Stride	NA	26	57	5	26	57	5	26
Random + Stride	NA	26	23	29	27	24	30	28
Random + No Stride	NA	26	23	29	31	29	34	40

B. Temporal Pointer Access Patterns

Pointer operations have been known to result in hard-to-predict random memory accesses that evade traditional stride-based load address prediction mechanisms. However, we find in this work that, most code regions access only a limited set of dynamically allocated buffers in any given execution interval (of 100 million dynamic instructions in our experiments), and often access these buffers in a deterministic sequence (one buffer after another), although the memory access patterns while accessing any one buffer may be fairly random. We identify several interesting patterns that are succinctly summarized in Table II. The first and the most trivial case is where one buffer is repeatedly accessed over a long execution interval, found extensively in the SPEC benchmarks *sjeng* and *lbm*. The second pattern is that of a striding access, where one buffer is accessed after another in the sequence of their allocation, in a striding pattern. These accesses may occur in batches, as shown in Listing 1, where each batch corresponds to a set of dereferencing operations (pointer chasing in this example) performed on any one buffer. The third pattern is that of a repeating access, where buffers are accessed in sequence (potentially with a stride), and that sequence repeats, as shown in Listing 2. Both these patterns occur extensively in all the

```
void chase(BUF* buf) {
    while (buf) {
       *buf = *buf + 10;
       buf = buf->next;
    }
}
while (true) {
    chase(buf11); // repeat PID(11)
    chase(buf15); // repeat PID(15)
    chase(buf19); // repeat PID(15)
}
```

Listing 1: Example Temporal Access Pattern: Batch + Stride

```
void update(BUF* buf) {
   *buf = *buf + 10;
   buf = buf->next;
}
while (true) {
    update(buf26);
    update(buf27);
    update(buf28);
} // repeat (PID(26), PID(27), PID (28))
```

Listing 2: Example Temporal Access Pattern: Repeat + Stride

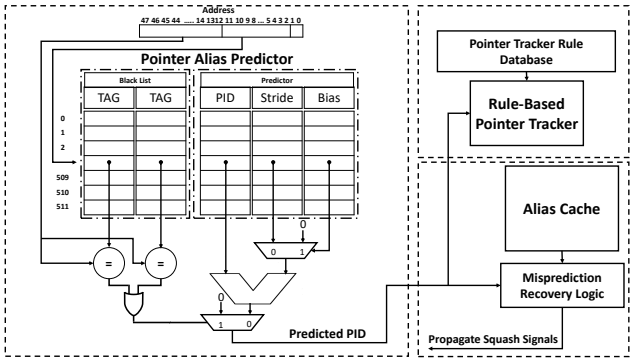


Fig. 4: Alias Detection in CHEx86

SPEC benchmarks we examine, with *perlbench* exhibiting the highest number of *"Batch + Stride"* patterns. Finally, buffers are sometimes accessed in random order of their allocation, but even in those instances, we observe striding access patterns.

These patterns are remarkably predictable as they are associated with the instruction address rather than the load effective address, allowing us to leverage and re-purpose conventional address prediction techniques [5] for fast detection of pointer reload operations at the front-end, as we switch from buffer to buffer. In the next two sections, we describe the implementation of such a predictor, along with misprediction recovery mechanisms.

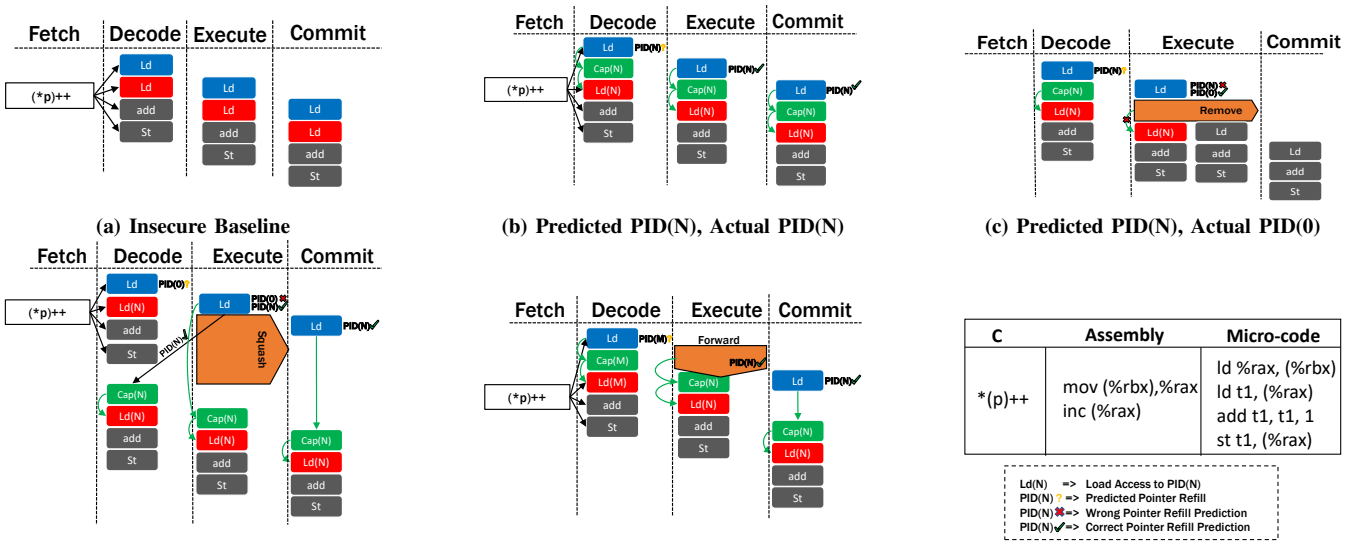
C. Spilled Pointer Alias Detection

One of the major advantages of our microcode-level protection mechanism is that it is context-sensitive, which allows us to only protect sensitive code regions and the data they access, as a result of which, instrumentation may only be surgically enabled for specific load/store operations. However, this involves identifying, at the front-end of the pipeline, what allocated buffer any given load/store instruction is accessing, so that an appropriate capability check may be injected into the decoded micro-op stream. Although it is possible to maintain a shadow table of spilled pointer aliases to store PID tags (albeit, only the finalized PID field) of spilled registers, it can only be looked up at the *execute* stage of the pipeline, when the effective address of a load instruction is available, significantly undermining our ability to detect instances of a spilled pointer getting reloaded to a register, at the front-end of the pipeline where we perform our microcode customization.

To ensure the seamless functioning of our pointer tracker, we develop a novel high-performance alias detection strategy that effectively predicts spilled pointer reload instances at the front-end of the pipeline. This is similar in spirit to load address prediction [5], but instead of predicting the effective address for every load instruction, we predict the PID that corresponds to the spilled pointer, potentially being reloaded into a register. Moreover, since pointer reload instances are fairly infrequent (for the SPEC CPU2017 benchmarks, only 2.5% of the memory references are spilled pointer reloads), the amount of predictor state required to perform pointer reload prediction is significantly lower than what is required for load address prediction. In fact, in this work, we show that even a simple 512-entry stride-based pointer reload predictor with 2-bit saturating counters (shown in Figure 4) can provide substantially high accuracy. We also employ a blacklist [32] of non-pointer reload operations to avoid destructive aliasing with other load instructions in the program that are loading data values rather than spilled pointer aliases. We note that such a high-performance alias detection feature not only significantly enhances our approach, but may potentially benefit other contemporary capability-based addressing systems.

Furthermore, just like any other speculative feature in the processor, it is imperative that we provide necessary mechanisms for misprediction detection and recovery, to ensure hazard-free execution. Since the effective address of a load is available at the *execute* stage of the pipeline, we are able to successfully perform a shadow alias table lookup to validate our predictions, as shown in Figure 4. There can be three types of pointer reload mispredictions: (1) *PNA0* (shown in Figure 5(c)) – where we predicted that we are reloading a pointer to a buffer with PID(N), but it was actually a buffer that we are not tracking (e.g., a stack buffer), in which case, we simply mark the injected capability check as an x86 *zero-idiom* that gets squashed at the instruction queue, before getting dispatched (similar to how NOPs get evaluated in Intel architectures), (2) *P0AN* (shown in Figure 5(d)) – where we are actually reloading a pointer to a buffer with PID(N), but we predicted that it is a buffer that we are not tracking, in which case, we leverage existing misprediction recovery mechanisms to flush the pipeline and restart execution at the offending instruction, with the right capability checks injected, and (3) *PMAN* (shown in Figure 5(e)) where we predicted that we are reloading a pointer to a buffer with PID(M), but it is a pointer to a buffer with PID(N), in which case, we simply forward the right PID and update our pointer tracking structures.

Misprediction detection is contingent on looking up a shadow table of spilled pointer aliases. To avoid the overhead of accessing this large shadow table for every load instruction, we employ several optimizations. First, we maintain a 256-entry 2-way set-associative in-processor alias cache, augmented by a 32-entry victim cache, to accelerate the lookup of aliases that are frequently spilled and reloaded. Note that as soon as the effective address of the load is available, we initiate the memory access for the load operation, and at the same time, also perform the alias cache access. Thus, the alias



(a) Insecure Baseline

(b) Predicted PID(N), Actual PID(N)

(c) Predicted PID(N), Actual PID(0)

(d) Predicted PID(0), Actual PID(N)

(e) Predicted PID(M), Actual PID(N)

(f) C, Assembly, and Microcode

Fig. 5: Pointer Reload Misprediction Recovery Mechanisms

cache access is not on the critical path of the load and does not impact the load-to-use latency. Second, since pointer reloads make up a small percentage of all memory references, we extend the metadata bits in the TLB and the page tables to include an *alias-hosting* bit that indicates if a page contains a spilled pointer, to further minimize the number of lookups. Third, we only allow committed store instructions to update the alias cache, to avoid transient stores along mispredicted paths from polluting it. Instead, for transient stores that may spill pointers to memory, we extend the store buffer to hold their corresponding PIDs, until the time they commit. Finally, similar to the in-memory page table structure, we implement a 5-level hierarchical shadow alias table structure that can be traversed using a hardware alias table walker, minimizing both the storage overhead and the latency of traversal. Unlike page table entries that hold the physical page number, an entry in the lowest-level alias table holds the PID of the pointer alias spilled at that virtual address. Note that the bounds and permissions information associated with that PID can be retrieved by a *capCheck* micro-op from a separate capabilities table.

In multithreaded environments, when a store instruction updates a spilled pointer alias on one core, invalidate requests are sent to all other cores in the system, to ensure that the in-processor alias caches in the system are coherent. The overheads due to this (including the overheads due to coherence misses in an alias cache) are modeled in all our multithreaded experiments.

D. Misspeculation Recovery Mechanisms

Our pointer tracking scheme may be prone to errors that manifest as a result of incorrect branch speculation and/or alias misprediction, since this tracking occurs entirely within the front-end. However, owing to the already well-established misprediction detection and recovery schemes implemented in modern processors, we can efficiently recover from misspeculation by simply extending a register's PID tags to include both transient and committed information, in which case, when a misprediction is detected, squash signals from the *execute* stage are propagated to the speculative pointer tracker, so that it can discard any incorrect transient execution state.

In particular, we extend the register tags maintained by the speculative pointer tracker to include two major fields – (1) *the finalized PID* propagated by the last committed instruction, and (2) *a vector of transient PIDs* containing the PIDs propagated by in-flight older instructions, along with their sequence numbers. When the pointer tracker receives a squash signal, just like the rest of the recovery logic in the pipeline, it inspects the sequence number of the offending instruction, and for each architectural register it tracks, it removes all transient PIDs that are associated with a sequence number that is greater than that of the offending instruction, from its tag. Furthermore, since the fetch stage is always ahead of the rest of the pipeline, the pointer tracker ensures that capability transfers are performed using the transient PID that is associated with the highest sequence number. While this approach limits the number of intrusive changes we make to the core design of the pipeline, we note that a more efficient implementation is possible by augmenting the register renaming logic [28], [60].

TABLE III
HARDWARE CONFIGURATION OF THE SIMULATED SYSTEM

Baseline Processor			
Frequency	3.4 GHz	I cache	32 KB, 8 way
Fetch width	4 fused uops	D cache	32 KB, 8 way
Issue width	6 unfused uops	ROB size	224 entries
INT/FP Regfile	180/168 regs	IQ	64 entries
RAS size	64 entries	BTB size	4096 entries
LQ/SQ size	72/56 entries	Functional Units	Int ALU (6) / Mult (1), FPALU (3) / SIMD (3)
Branch Predictor	LTAGE		

VI. Methodology

Baseline Architecture. We use the Gem5 [8] architectural simulator to model both our CHEx86 architecture and the insecure baseline x86 architecture. Both architectures are modeled after Intel Skylake. Table III describes their detailed architectural configuration. Further, we also evaluate CHEx86 with different capability and alias cache configurations.

Benchmark Selection. To characterize the performance impact due to our approach, we use the C and C++ benchmarks from the SPEC CPU2017 [55] and PARSEC 2.1 [7] suite. We also use PinPlay [56] and Simpoint [59], [66] tools to select representative regions for simulation. These benchmarks are compiled at the -O3 optimization level using the LLVM/-Clang [41], [42] compiler infrastructure for comparison with LLVM's address sanitizer.

Security Evaluation. To measure the security effectiveness of our approach, we use three exploit suites: (a) Runtime Intrusion Prevention Evaluator [83] that comprises of 850 exploits targeting spatial safety violations, generated by sweeping across five different dimensions (location of the buffer, target code pointer, direct/indirect buffer overflow, type of shellcode, and the libc being function abused), (b) LLVM's address sanitizer [65] test suite that comprises of unit test cases that test the ability of the address sanitizer to flag typical memory safety violations, and (c) How2Heap from the CTF team ShellPhish that comprises of 18 distinct evasive exploits that corrupt the heap metadata by targeting a variety of spatial and temporal safety violations.

VII. Results

In this section, we first discuss the effectiveness of CHEx86 in mitigating memory safety violations, and then present results from a detailed performance evaluation.

A. Security Evaluation

We note that CHEx86 is able to successfully thwart all exploits from RIPE [83], LLVM's ASan [65], and ShellPhish's How2Heap, completely under-the-hood, without making any modifications to the source/object code, while having access to limited source-level symbol information. Regardless of the degree of evasion used to trick the memory allocator, the principal anchor points for most exploits remain buffer *out-of-bounds* accesses, *use-after-free*, *illegal free*, and *double free*, with the exception of two ASan exploits that attempt a resource exhaustion attack by allocating prohibitively large memory blocks. The rest of this section will describe in detail how CHEx86 mitigates each of these violations.

Out-Of-Bounds Accesses. Recall from Section IV that CHEx86 instruments all pointer dereferencing operations with a special *capCheck* micro-op that looks up the capability cache (or the shadow capability table, in case of a miss) with the appropriate PID, to check if the address being dereferenced lies within the bounds maintained in the capability, and further flags violations. In our experiments, all RIPE exploits, four exploits from ASan, and six from How2Heap are flagged for out-of-bounds accesses by CHEx86, regardless of the method used by the attacker to perform these violations.

Use-After-Free. In addition to validating bounds information, the *capCheck* micro-op also validates permissions, including checking the *valid* bit. If the *valid* bit is set to zero, it indicates access to a freed memory block, and thus flagged as a violation. While none of the RIPE exploits, and only two ASan exploits (*tail magic* and *UAF with RB Distance*) target *use-after-free*, a majority of the How2Heap exploits corrupt heap metadata through a *use-after-free* violation.

Invalid Free and Double Free. Again, recall from Section IV that CHEx86 intercepts both entry and exit points of registered heap management functions, including *free*, and upon intercepting the entry point of the *free* function, a *capFree.begin* micro-op is injected (with the appropriate PID) into the execution stream, to initiate the *capability free* process. However, if the *capFree* micro-op finds that the PID is zero or that the *valid* bit has already been set to zero, it immediately throws an *invalid free* or a *double free* exception respectively, to prevent further memory corruption. In our experiments, we detect seven violations due to *invalid free* and/or *double free*, with six of these coming from How2Heap and one from ASan.

Heap Spraying and Resource Exhaustion. CHEx86 is also capable of defending against heap spraying and resource exhaustion attacks that hinge on allocating large blocks of memory. Two exploits from ASan (*allocator returns NULL* and *sizes*) are representative anchor points for such attacks. We are able to detect such violations via the *capGen.begin* micro-op at the time of capability generation, by checking if the request size of allocation lies within the pre-configured maximum allocatable size for any given memory block (1GB in our experiments).

B. Discussion on False Positives

Intentional Constant Dereferencing. In many x86-64 applications, global data structures are intentionally accessed by dereferencing constant integer addresses. There are two primary ways in which this is done. In the first and most common case, these addresses are retrieved from a constant pool within the text section using a PC-relative load instruction, allowing us to automatically track the global address being loaded into the register, triggering appropriate capability checks. Second, in some applications that are statically linked against *libstdc++*, we observe instances of constant integer addresses being directly moved into a register and then subsequently dereferenced using the register. As noted in Section V-A, our pointer tracker flags such wild dereferences as potential capability violations, resulting in false positives. We observe one instance of such a false positive when the benchmark *leela* is statically linked against *libstdc++*.

Non-Local Control Transfers. In C++ applications, non-local control transfers may occur as a result of (a) exceptions – in which case, the stack is unwound frame-by-frame, cleaning up any heap-allocated buffers along the way, and (b) setjmp/-longjmp – where the calling context is restored using a jump buffer, albeit without cleaning up heap-allocated buffers. In both cases, we observe that pointer aliases that are spilled to memory are reloaded back into registers during a context restore, and therefore, we do not observe any false positives or false negatives.

C. Comparison with Prior Approaches

Table IV compares CHEx86 with other prior memory safety techniques in terms of performance, security, storage overhead, binary compatibility, and hardware complexity. While the core functionality of CHERI and CHEx86 is similar in that they dynamically perform capability checks in hardware for every pointer dereferencing operation, CHERI requires recompilation unlike CHEx86, and suffers from high storage overhead for pointer-intensive applications due to its tagged memory implementation. Out of the other schemes, BOGO [92], REST [68], Califorms [63], and Watchdog [49] are the only

TABLE IV
COMPARISON WITH PRIOR MEMORY SAFETY TECHNIQUES

Proposal	Temporal Safety	Spatial Safety	Metadata	Binary Compatibility	Performance(%) (Average) (Bench)	Storage Overhead(%) (Average) (Bench)	Hardware Modifications
Hardbound [24]	✘	✓	Shadow	Partial	5% (Olden)	55% (Olden)	Tag metadata cache + TLB, μop injection logic
Watchdog [49]	✓	✓	Shadow	Partial	24% (SEPC2000)	56% (SPEC2000)	Renaming logic, μop injection logic Lock location cache
Intel MPX [54]	✘	✓	Inline	✘	80% (SEPC2006)	150% (SPEC2006)	N/A
BOGO [92]	✓	✓	Inline	✘	60% (SEPC2006)	36% (SPEC2006)	N/A
CHERI [86]	✘	✓	Inline	✘	18% (Olden)	90% (Olden)	Capability coprocessor, Tag cache Capability Unit
CHERIvoke [87]	✓	✘	Inline	✘	4.7% (SPEC2006)	12.5% (SPEC2006)	Capability co-processor, Tag cache Tag controller, Capability unit
REST [68]	✓	✓	Shadow	✘	23% (SPEC2006)	N/A	1-8b per L1D line, 1 comparator
Califorms [63]	✓	✓	Shadow	✘	16% (SPEC2006)	N/A	8b per L1D line, 1b per L2/L3 line
CHEx86	✓	✓	Shadow	✓	14% (SPEC2017)	38% (SPEC2017)	μop injection logic, Capability$, Alias$ Speculative Pointer Tracker

other works apart from CHEx86 that offer both temporal and spatial memory safety protections. BOGO is a software solution that extends Intel's MPX to provide temporal memory safety, but incurs severe performance degradation for many applications. The tripwire-based approaches, REST [68] and Califorms [63], are attractive in terms of performance and low hardware overhead, but they necessarily require recompilation. While Watchdog also dynamically instruments the micro-op stream with temporal and spatial memory safety checks, CHEx86 differs from Watchdog in several important ways.

First, Watchdog conservatively instruments every 64-bit integer load/store instruction in the program with memory safety checks, as opposed to the more targeted and prediction-driven instrumentation in CHEx86. Our experiments on the SPEC CPU2017 benchmarks indicate that such a conservative scheme could impact the overall execution time by 40% on average, slowing down pointer-intensive applications such as *xalancbmk* by as much as 2X. To mitigate this severe degradation in performance, Watchdog suggests extending the ISA and the compiler to annotate pointer dereferencing operations. This not only necessarily requires recompilation or binary translation, but entails substantial development, verification, porting, and deployment costs.

Second, a key distinguishing feature of CHEx86 is that it has the ability to speculatively detect spilled pointer aliases getting reloaded into registers, without having to lookup any metadata in shadow memory. Watchdog, on the other hand, deals with spilled pointer aliases by associating every word in memory with pointer identifier metadata in the shadow memory – when a pointer is reloaded to a register, the associated identifier metadata is also read from shadow memory, increasing the number of memory references by as much as 2X, thereby negatively impacting both performance and energy efficiency. We note that the shadow memory overhead in CHEx86 scales by the number of allocations (capabilities table) and the number of references (alias table), rather than by the number of words in memory. In particular, we observe an average reduction of 32% in shadow memory overhead, in comparison to Watchdog.

Third, the pointer tracking scheme in Watchdog is significantly less flexible than that of CHEx86, as it is tightly integrated with the register renaming logic. In contrast, the pointer tracking logic and the rule database in CHEx86 is completely decoupled from register renaming and other core logic on the decode/rename critical path, allowing us for example, to update pointer tracking rules on-demand and in the field using microcode updates.

Finally, we present a more complete and flexible scheme that can be integrated with modern x86 processors with minimal intrusion, along with an extensive security analysis that covers a wide range of evasive modern-day memory safety exploits.

D. Performance Evaluation

Figure 6 (top) shows the performance degradation due to CHEx86, in terms of the overall execution time. Clearly, from the graph, we sacrifice little performance in comparison to the software-based address sanitizer for most benchmarks. The prediction-driven microcode-level variant of CHEx86 always outperforms the always-on strategy that instruments every load/store instruction, regardless of whether it performs a heap access. It also only slightly trails behind the hardware-only variant, and supersedes it for the memory-intensive applications *leela*, *mcf*, and *xalancbmk*. This is because the hardware-only scheme directly affects the performance of all loads and stores in the program. Moreover, it consistently outperforms the binary translation-driven variant and in fact, by moving the secure instrumentation functionality from the binary translator to the microcode engine, we significantly enhance the front-end throughput, and speed up execution overall, by 12%, on average. In comparison to a software-only defense, the microcode-based variant of CHEx86 speeds up execution by 59% for SPEC and 2.2X for PARSEC, and with respect to an insecure baseline architecture that is vulnerable to memory corruption exploits, it slows down execution by an average of 14% for SPEC, and 9% for PARSEC. Note that this is heavily dominated by outliers such as *mcf* and *xalancbmk* that perform pointer-intensive computation for most of their execution.

Overall, surgical on-demand instrumentation of the micro-op stream with capability checks offers significant advantages over the other variants. First, it always offers better performance than the binary translation variant and the software-based address sanitizer, while maintaining software compatibility. Second, it only slightly trails behind the hardware-only scheme, while being less intrusive – importantly, it does not change the behavior of loads and stores, and performs all instrumentation using existing micro-ops.

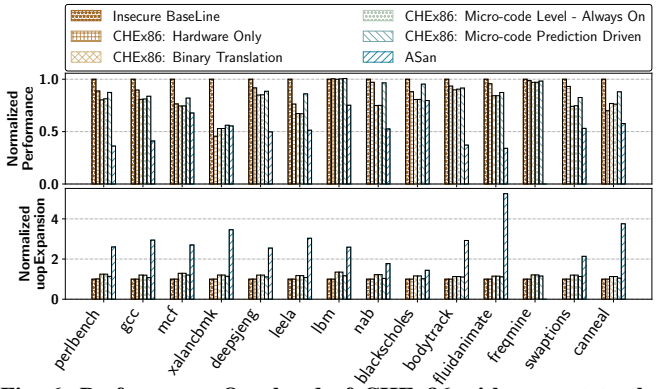


Fig. 6: Performance Overhead of CHEx86 with respect to that of an insecure baseline and LLVM's Address Sanitizer

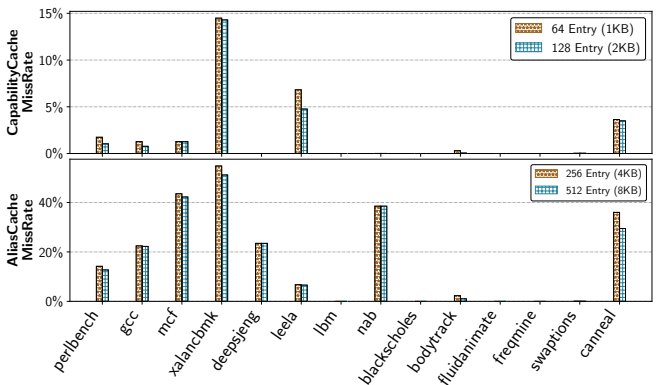


Fig. 7: Capability (top) and Alias Cache (bottom) Miss Rates

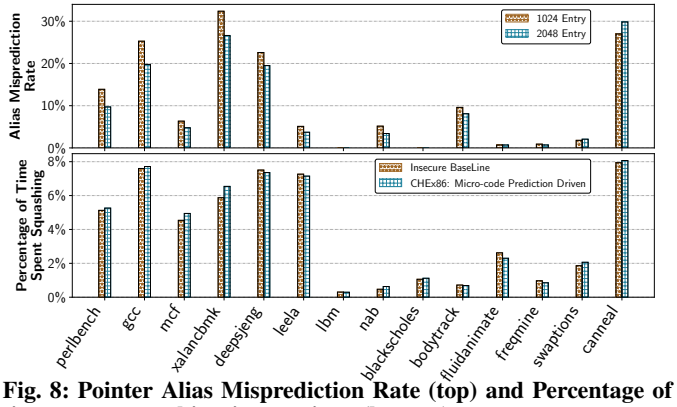


Fig. 8: Pointer Alias Misprediction Rate (top) and Percentage of time spent squashing instructions (bottom)

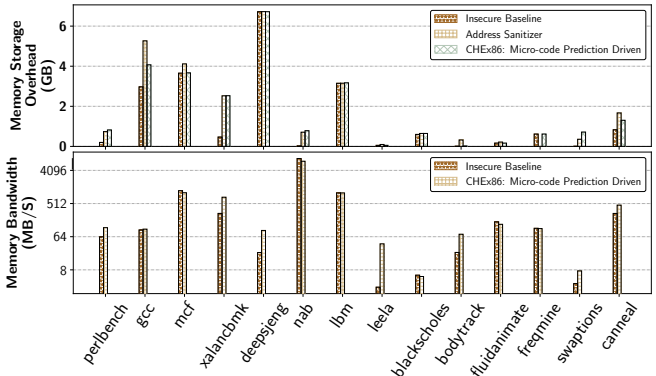


Fig. 9: Memory Storage Overhead (top) and Memory Bandwidth Impact (bottom)

We next perform a detailed technical evaluation to understand how the different components of CHEx86 contribute to the 14% slowdown with respect to the insecure baseline. The primary factor that affects performance in CHEx86 is the micro-op instrumentation for generating, checking, and freeing capabilities, considerably expanding the number of dynamic micro-op instructions executed by the processor. Figure 6 (bottom) shows the increase in the number of dynamic micro-ops executed due to both the software-based instrumentation of the address sanitizer and the microcode-level instrumentation in CHEx86. We incur an average micro-op expansion of 11% (for SPEC and PARSEC) in comparison to the address sanitizer that more than doubles the number of dynamic instructions executed. The amount of micro-op expansion is directly proportional to the extent of pointer activity in an application – for code regions without significant pointer activity, we offer close to native performance. In fact, in a context-sensitive implementation, where only security-critical code regions are instrumented, we track all allocations, but inject *capCheck* micro-ops for only those pointer dereferencing instructions that lie in security-critical code, greatly reducing the micro-op bloat.

Furthermore, to provide capability-based protection, completely under-the-hood, without sacrificing binary compatibility, we maintain all our capability metadata in shadow tables, and avoid frequent trips to memory by leveraging a small in-processor capability cache. Figure 7 (top) shows the miss rate of our 64-entry in-processor capability cache. Recall from Section IV that the number of allocations in use at any given point in time is significantly lower than the total number of allocations a process makes during its lifetime, which allows us to efficiently track in-use capabilities using a capability cache holding only 64 capabilities, with a miss rate of only 2.1% on average.

We next evaluate the impact of the alias cache on performance. Again recall from Section V that pointer reloads are infrequent, and can therefore be efficiently tracked using a small 2-way set associative 256-entry alias cache that is augmented by a 32-entry fully associative victim cache. Clearly from Figure 7 (bottom), our alias cache miss rate is negligible for most applications, resulting in very few extra trips to memory for looking up aliases. Our alias cache incurs an average miss rate of 17.3% across the SPEC and PARSEC benchmarks, again, heavily dominated by outliers.

In Figure 8 (top), we show the misprediction rate of the pointer alias detection unit. Again, given the rarity of pointer reloads, we are able to predict these events with high accuracy (89% on average for SPEC and PARSEC) using a simple stride-based prediction scheme. We note that with this high level of accuracy, comes the knowledge of whether a given instruction reloads a pointer from memory, and more importantly the PID associated with the spilled pointer alias. Figure 8 (bottom) compares the percentage of time spent squashing in both the insecure baseline and CHEx86. We observe that this number only slightly increased, indicating that the contribution of the pointer alias misprediction squash penalty to the overall performance degradation is negligible, in comparison to other factors such as micro-op expansion.

Finally, we evaluate the impact of our shadow tables on memory bandwidth and the memory storage. Figure 9 (bottom) compares the bandwidth usage of our applications on both CHEx86 and the insecure baseline x86 architecture. Due to our low miss rates for both the capability cache and the alias cache, we do not observe any significant change in the memory bandwidth usage. Outliers include *xalancbmk*, *leela* and *deepsjeng* that are characterized by notably intense pointer activity. However, we note that, even for these benchmarks, the bandwidth usage is contained at an acceptable limit. Further, in the same Figure (top), we also show the memory storage overhead of CHEx86, by examining the increase in an application's resident set size. We also include the address sanitizer in our comparison since it also uses shadow memory for providing security. We note that we do not allocate any more shadow memory than the address sanitizer, while performing significantly better.

Overall, we provide effective high performance microcode-level mitigations against several attack vectors that exploit temporal and spatial memory safety violations, on unmodified source and object code, completely under-the-hood.

VIII. Conclusions

This work proposes the CHEx86 processor architecture for safeguarding applications, including legacy binaries, against a multitude of security exploits that target common temporal and spatial memory safety vulnerabilities by instrumenting the code at the microcode-level, completely under-the-hood. The novel extensions we propose include: (a) microcode-level capability enforcement, (b) a fully automated speculative pointer tracking scheme, and (c) an efficient scheme for detecting pointer aliases in memory. Overall, we incur 14% degradation in performance, while outperforming a state-of-the-art software-based mitigation by 59%. This is also the first work to identify the temporal access patterns of pointer accesses, and demonstrate that these can be effectively predicted using a simple stride-based prediction mechanism.

IX. Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful insights. This research was supported by NSF Grant CNS-1850436, NSF/Intel Foundational Microarchitecture Research Grant CCF-1912608, and a DARPA contract under the agreement number HR0011-18-C-0020.

References

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*, 2009.

[2] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation." in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[3] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach, "A verified information-flow architecture," in *POPL*, 2014.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *SOSP*, 2009.

[5] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. C. Weiser, "Correlated load-address predictors," in *ISCA*, 1999.

[6] E. D. Berger and B. G. Zorn, "Diehard: probabilistic memory safety for unsafe languages," in *PLDI*, 2006.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*, 2008.

[8] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *MICRO*, 2006.

[9] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *ISSTA*, 2012.

[10] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *ASPLOS*, 1994.

[11] H. Chen, X. Wu, L. Yuan, B. Zang, P. c. Yew, and F. T. Chong, "From speculation to security: Practical and efficient information flow tracking using speculative hardware," in *ISCA*, 2008.

[12] S. Chen, P. B. Gibbons, M. Kozuch, and T. C. Mowry, "Log-based architectures: Using multicore to help software behave correctly," *OSR*, 2011.

[13] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, "Software-based gate-level information flow security for iot systems," in *MICRO*, 2017.

[14] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. Son *et al.*, "Cheri jni: Sinking the java security model into the c," in *ASPLOS*, 2017.

[15] M. L. Corliss, E. C. Lewis, and A. Roth, "Dise: A programmable macro engine for customizing applications," in *ISCA*, 2003.

[16] M. L. Corliss, E. C. Lewis, and A. Roth, "Low-overhead interactive debugging via dynamic instrumentation with DISE," in *HPCA*, 2005.

[17] M. L. Corliss, E. C. Lewis, and A. Roth, "Using dise to protect return addresses from attack," *WASSA*, 2005.

[18] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting systems from stack smashing attacks with StackGuard," in *Linux Expo*, 1999.

[19] C. Cowan, C. Pu, D. Maier *et al.*, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security Symposium*, 1998.

[20] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *MICRO*, 2004.

[21] K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic, "Talx86: A realistic typed assembly language," in *Workshop on Compiler Support for System Software Atlanta*, 1999.

[22] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *ISCA*, 2007.

[23] B. Davis, R. N. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka *et al.*, "Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the posix c run-time environment," in *ASPLOS*, 2019.

[24] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," in *ASPLOS*, 2008.

[25] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *IMC*, 2014.

[26] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Heaphopper: Bringing bounded model checking to heap implementation security," in *USENIX Security Symposium*, 2018.

[27] H. Etoh, "GCC extension for protecting applications from stack-smashing attacks," 2003. [Online]. Available: http://www.research.ibm.com/trl/projects/security/ssp

[28] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta, "Continuous optimization," in *ISCA*, 2005.

[29] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The m-machine multicomputer," *International Journal of Parallel Programming*, 1997.

[30] B. Hawkes, "0day "in the wild"," 2019.

[31] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in *USENIX Security Symposium*, 2018.

[32] Y. Ishii, K. Kuroyanagi, T. Sawada, M. Inaba, and K. Hiraki, "Revisiting local history to improve the fused two-level branch predictor," *3rd Championship Branch Prediction*, 2010.

[33] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c." in *ATC*, 2002.

[34] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. Watson, D. Chisnall, M. Roe, B. Davis *et al.*, "Efficient tagged memory," in *ICCD*, 2017.

[35] R. W. Jones and P. H. Kelly, "Backwards-compatible bounds checking for arrays and pointers in c programs," in *International Workshop on Automatic Debugging*, 1997.

[36] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *SOSP*, 2009.

[37] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *S&P*, 2018.

[38] B. Kollenda, P. Koppe, M. Fyrbiak, C. Kison, C. Paar, and T. Holz, "An exploratory analysis of microcode as a building block for system defenses," in *CCS*, 2018.

[39] P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar, and T. Holz, "Reverse engineering x86 processor microcode," in *USENIX Security Symposium*, 2017.

[40] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *CCS*, 2013.

[41] C. Lattner, "LLVM and Clang: Next Generation Compiler Technology," in *The BSD Conference*, 2008.

[42] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO*, 2004.

[43] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification." in *NDSS*, 2015.

[44] Z. Liu and J. Criswell, "Flexible and efficient memory object metadata," *ISMM*, 2017.

[45] J. McMahan, M. Christensen, K. Dewey, B. Hardekopf, and T. Sherwood, "Bouncer: Static program analysis in hardware," *ISCA*, 2019.

[46] J. McMahan, M. Christensen, L. Nichols, J. Roesch, S.-Y. Guo, B. Hardekopf, and T. Sherwood, "An architecture supporting formal and compositional binary analysis," *OSR*, 2017.

[47] M. Miller, "Trends and challenges in the vulnerability mitigation landscape." USENIX Association, 2019.

[48] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system f to typed assembly language," *TOPLAS*, 1999.

[49] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *ISCA*, 2012.

[50] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *PLDI*, 2009.

[51] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," in *TOPLAS*, 2002.

[52] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.

[53] G. Novark and E. D. Berger, "Dieharder: securing the heap," in *CCS*, 2010.

[54] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: An empirical study of Intel MPX and software-based bounds checking approaches," *arXiv:1702.00719*, 2017.

[55] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a decade: Did SPEC CPU 2017 broaden the performance horizon?" in *HPCA*, 2018.

[56] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *CGO*, 2010.

[57] PaX Team, "PaX address space layout randomization," 2003. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[58] PaX Team, "PaX non-executable pages design and implementation," 2003. [Online]. Available: http://pax.grsecurity.net/docs/noexec.txt

[59] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for Accurate and Efficient Simulation," in *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2003.

[60] V. Petric, T. Sha, and A. Roth, "Reno: a rename-based instruction optimizer," in *ISCA*, 2005.

[61] F. Qin, S. Lu, and Y. Zhou, "Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs," in *HPCA*, 2005.

[62] D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heap exploits," in *PLAS*, 2017.

[63] H. Sasaki, M. A. Arroyo, M. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical byte-granular memory blacklisting using califorms," *MICRO*, 2019.

[64] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy." in *USENIX Security Symposium*, 2011.

[65] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *ATC*, 2012.

[66] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *ASPLOS*, 2002.

[67] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "Freeguard: A faster secure heap allocator," in *CCS*, 2017.

[68] K. Sinha and S. Sethumadhavan, "Practical memory safety with rest," in *ISCA*, 2018.

[69] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ASPLOS*, 2004.

[70] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A single-chip secure processor," *Information Security Technical Report*, 2005.

[71] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and implementation of the aegis single-chip secure processor using physical random functions," in *ISCA*, 2005.

[72] M. Taram, D. Tullsen, A. Venkat, H. Sayadi, H. Wang, S. Manoj, and H. Homayoun, "Fast and efficient deployment of security defenses via context sensitive decoding," in *GOMACTech*, Mar 2019.

[73] M. Taram, A. Venkat, and D. Tullsen, "Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency," in *ISCA*, 2018.

[74] M. Taram, A. Venkat, and D. M. Tullsen, "Context-sensitive decoding: On-demand microcode customization for security and energy management," *MICRO*, 2019.

[75] M. Taram, A. Venkat, and D. M. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *ASPLOS*, 2019.

[76] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ASPLOS*, 2009.

[77] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "Rifle: An architectural framework for user-centric information-flow security," in *MICRO*, 2004.

[78] A. Venkat, A. Krishnaswamy, K. Yamada, and R. Palanivel, "Binary Translation driven Program State Relocation," in *United States Patent Grant US009135435B2*, 2015.

[79] A. Venkat, S. Shamasunder, H. Shacham, and D. M. Tullsen, "Hipstr: Heterogeneous-isa program state relocation," in *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, 2016.

[80] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, Feb 2008, pp. 173–184.

[81] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *S&P*, 2015.

[82] D. Weston and M. Miller, "Windows 10 mitigation improvements," *Black Hat USA*, 2016.

[83] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "Ripe: runtime intrusion prevention evaluator," in *ACSAC*, 2011.

[84] M. V. Wilkes and R. M. Needham, "The cambridge cap computer and its operating system," 1979.

[85] J. Woodruff, A. Joannou, H. Xia, B. Davis, P. G. Neumann, R. N. M. Watson, S. Moore, A. Fox, R. Norton, and D. Chisnall, "Cheri concentrate: Practical compressed capabilities," *IEEE Transactions on Computers*, 2019.

[86] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *ISCA*, 2014.

[87] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. Watson *et al.*, "Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety," in *MICRO*, 2019.

[88] J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high performance computing." in *NDSS*, 2019.

[89] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data," in *MICRO*, 2019.

[90] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," *arXiv:1903.00503*, 2019.

[91] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *ASPLOS*, 2015.

[92] T. Zhang, D. Lee, and C. Jung, "Bogo: Buy spatial memory safety, get temporal memory safety (almost) free," in *ASPLOS*, 2019.