MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation

Lucian Mogosanu[1], Ashay Rane[2], and Nathan Dautenhahn[3]

[1] University POLITEHNICA of Bucharest–lucian.mogosanu@cs.pub.ro
[2] The University of Texas at Austin–ashay@cs.utexas.edu
[3] Rice University–ndd@rice.edu

Abstract. In this work we present, MicroStache, a specialized hardware mechanism and new process abstraction for accelerating *safe region* security solutions. In the *safe region* paradigm, an application is split into safe and unsafe parts. Unfortunately, frequent mixing of safe and unsafe operations stresses memory isolation mechanisms. MicroStache addresses this challenge by adding an orthogonal execution domain into the process abstraction, consisting of a memory segment and minimal instruction set. Unlike alternative hardware, MicroStache implements a simple microarchitectural memory segmentation scheme while integrating it with paging, and also extends the *safe region* abstraction to isolate data in the processor cache, allowing it to protect against cache side channel attacks. A prototype is presented that demonstrates how to automatically leverage MicroStache to enforce security polices, SafeStack and CPI, with 5% and 1.2% overhead beyond randomized isolation. Despite specialization, MicroStache enhances a growing and critical programming paradigm with minimal hardware complexity.

Keywords: intra-process isolation, safe region, security microarchitecture

1 Introduction

Computing systems hold a significant amount of personal data. Unfortunately, applications are subject to memory safety violations that allow attackers access to application data or to take over the system. A popular and well explored solution is to provide full memory safety, which forces all data access to be safe (*e.g.,* eliminating writing outside the bounds of an object) [12,15,35,40,34]. Despite solving the problem, full memory safety comes with too high a price, hindering its mainstream use. Instead, an emerging paradigm protects only sensitive program data, such as code pointers [30], cryptographic keys [23], or programmer-defined structures [8,13]. In general, partial memory safety places *sensitive* objects into isolated memory regions, called *safe regions* [28], that can only be accessed by privileged program instructions. The result is powerful security solutions (themselves eliminating large classes of exploits) at a fraction of the cost.

Despite the demonstrable rise and power of the *safe region* paradigm, commodity protection mechanisms are inadequate. This is because safe region isolation relies heavily on the interleaving between sensitive and regular memory accesses, leaving existing mechanisms with two options: either monitor all unsafe accesses (*e.g.,* sandboxing them using SFI or MPX[43,28,39,8]) or incur costly protection domain switches (as in SGX, TrustZone, MPK, VMFUNC [28,29]). Alternatively, tag-based architectures allow policy enforcement at instruction and word granularity [41,16,9], but require significant hardware enhancements.

The core research question we investigate in this work is how to most effectively support the *safe region* paradigm, specifically seeking a hardware accelerated abstraction and mechanism. Our goal is to do so in the most general and simplest way, so that the design may be portable to alternative architectures, and to efficiently isolate regions within an address space without requiring explicit monitoring of unsafe operations or domain switching.

To this end, we propose a novel memory abstraction that is isolated by a hardware data structure. MicroStache inserts an independently addressed memory region, the *stache segment*, into the standard process environment. The stache is accessed through a small instruction set extension that operates from within program context, as an embedded but orthogonal execution domain. The stache can be used to efficiently realize *safe region* solutions by controlling when and where stache access occurs. MicroStache also includes a hardware stack that not only supports standard stack behavior but also enforces a new stack safety property, where stack access is restricted to the currently executing frame. Furthermore, we extend the MicroStache abstraction into the microarchitecture to provide static cache separation and special mechanisms that can be leveraged by programmers for cache side channel defense.

To demonstrate our system, we show that it can be used for a variety of security applications that protect sensitive program data and user secrets either manually or through automated static checking. We implement two memory safety solutions (SafeStack and Code-Pointer Integrity [30]) and find that, relative to randomization based isolation, MicroStache incurs 5% and 1.2% overhead respectively. We demonstrate our claims through a prototype MicroStache implementation for x86-64 in the Gem5 [4] simulator, along with LLVM compiler support. Overall, our contributions include:

- The design and implementation of MicroStache, a novel abstraction for sensitive data isolation by confining it to a dedicated memory segment and separating memory accesses at the ISA level (Section 4).
- A framework for implementing arbitrary data protection policies for user applications, either manually or automatically (using compiler support) and a demonstration of this framework on several real-world scenarios (Section 5).
- MicroStache Safe Stack and an attack surface analysis that illustrates the security gained through its enforcement: elimination of 60–75 % of Turing-complete gadgets and mitigation of cache side channel attacks (Section 7).

2 Background and Motivation

The primary goal of MicroStache is to provide an abstraction for efficient and effective in-address space protection of *safe regions* [28]. Protection means preserving the integrity and confidentiality of *sensitive* data, (e.g. cryptographic keys, passwords, shadow structures and metadata used to implement partial memory safety) stored within the *safe region.* A secondary goal, is to use the abstraction as a means with which to specify and enforce data cache isolation. This section explores the degree to which existing mechanisms solve this issue. Table 1 compares software and hardware isolation mechanisms, including implementations available on commodity hardware and state of the art systems.

2.1 Safe Region Paradigm

At it's core the safe region paradigm places sensitive program data in a special memory region that is accessible only to a subset of the program's instructions. The use of this region depends on the security policy. For example, spatial memory safety approaches place object bound metadata into the safe region and verify that pointer dereferences are in bounds [12,15,35,40,34]. The approach requires that only the security runtime, which updates bounds metadata, is permitted to modify the safe region. In general, there are many security policies that employ the pattern of allocating sensitive data into the safe region and then protecting it from unprivileged access (see Koning *et al.* [28] for a thorough description of policies). Selecting which instructions are privileged is specific to the security policy and typically done by a static analysis tool, like a compiler. Additionally, these policies are best when performed "in context" because they require frequent access to the safe region.

2.2 Mechanisms

Once data and instructions are divided into privileged and unprivileged parts, a mechanism is needed to protect at runtime. Typically mechanisms use one of the following methods: 1) sandboxing, 2) separating, or 3) elevating privileges.

Sandboxing Instructions In the sandboxing approach, the safe region is allocated into the traditional process address space, accessible to regular instructions which become privileged by definition, and unprivileged instructions are explicitly constrained from accessing the safe region. Software Fault Isolation (SFI) uses an inline reference monitor that denies unprivileged access to the safe region by checking every access [43,39]. Unfortunately, SFI incurs relatively high overheads. Intel MPX, accelerates range checks by performing them in hardware [28,8], however, it suffers from costly bounds updating operations, must still explicitly monitor all unprivileged operations (which can be the majority), and is x86 specific.

Table 1. Comparison between data isolation mechanisms. **Op. Granularity**: operation granularity (smallest protected unit). **Iso Type**: type of isolation mechanism—sandboxing unprivileged instructions; separating into domains requiring context switches; or elevating privilege on a per instruction basis. **Comm.**—available in commodity systems. **HW Complexity**—complexity of the hardware support required. **Side Channel**—provides microarchitectural side channel defense. **Overhead**—run-time overhead: ranges based on the minimum and average values reported. [a]Nonderministic defense. [b]Depends on address mask. [c]Unavailable at time of writing.

Mechanism	**Op. Gran.**	**Iso Type**	**Comm.**	**HW Complexity**	**Side Channel**	**Overhead**
ASLR [30]	byte	elevate	✓	none	–	$\approx 0\,\%$[a]
SFI [39,30]	—[b]	sandbox	✓	none	–	low-high
Segments [46,30]	byte	elevate	✓	medium	–	low
MPX [8,28]	byte	sandbox	✓	medium	–	medium
MPK [28]	page	domain	✓	medium	–	low
VMFUNC [31,28]	page	domain	✓	high	–	medium
SGX [3,1]	page	domain	✓	high	–	high
TrustZone [2,24]	page	domain	✓	high	–	high
HDFI [41]	word	elevate	–	medium	–	low-high
PUMP [38]	byte	elevate	–	high	–	low-high
IMIX [19]	page	elevate	–	low	–	low
MicroStache	byte	elevate	–	low	✓	low

Separating into Domains Instead of restricting unprivileged instructions, *domain* based schemes place the safe region into an orthogonal area of memory requiring some form of context switch to access. In this way, each privileged operation must perform a domain switch, leaving regular instructions unchanged. Intel SGX, VMFUNC, and MPK, as well as ARM TrustZone provide domain switch isolation. However, all suffer from high overhead [28].

Elevating Instructions Both prior approaches are costly, requiring either frequent checks to sandbox unprivileged access or incur expensive domain switches. Elevation based approaches leave unprivileged instructions alone and add mechanisms to increase the privilege of the sensitive instructions. Randomization mechanisms place the safe region in a random location that only the privileged instructions know [30]. Despite their efficiency, randomization techniques are probabilistic, leading to exploits [18,22,21].

Split instruction set approaches create special instructions with privileges to directly access the safe region, while normal instructions are mediated by custom hardware. In x86-32 segmentation, the safe region is placed in a separate segment, and then only special segment-based instructions can access the region. Unfortunately, segmentation has disappeared from modern architectures and its microarchitecture is complex to support in general. IMIX, adds a new safe region page permission, which is only accessible through a new instruction [19]. This scheme is comparable to MicroStache, however, details have yet to be published.

Another way of elevating specific instructions is to use a tag-based microarchitecture. In tagging schemes, each instruction can be tagged with a policy denoting its privilege. Several tagging schemes have been proposed, but more recently region based schemes, HDFI [41] and the PUMP [38], have demonstrated the ability to enforce *safe region* based polices. Despite the powerful nature of these schemes, they require complex hardware that inhibits path to adoption. Moreover, partial memory safety techniques such as CPI [30] and DCI [8] are inherently dependent on the existence of metadata and safe regions, which to our knowledge HDFI cannot easily implement.

2.3 Extending the Safe Region to Cache Level Isolation

In addition to the safe use of memory, we seek to prevent leakage of secret information through cache side channels, which have been used to break confidentiality of both cryptographic and non-cryptographic applications. Both software as well as microarchitectural solutions exist, but not without drawbacks. Software solutions sidestep microarchitectural modifications [27], thus enabling defenses on commodity processors, but they also incur substantial overheads due to their reliance on generic ISA instructions. Microarchitecture-only solutions are generally faster but lack flexibility [44], because defenses ignore contextual information about the applications. MicroStache embodies a hardware-software design that is able to leverage the best of both worlds by extending trust to portions of the microarchitecture, while also leveraging the compiler to identify the parts of the program that need protection from cache side channels. Furthermore, the abstraction boundary provided by the safe region paradigm is similar to the types of protection desired against side channels. Doing so requires specialized hardware.

2.4 MicroStache Design Goals

From our analysis, we argue that in-place solutions offer the best in terms of programmability and efficiency, and identify the following design requirements that a *safe region* abstraction should meet:

Requirement 1 (Performance Symmetry)**.** *Regular* and *sensitive* memory accesses should impose the same performance penalty.

Requirement 2 (Programmability)**.** Isolation mechanisms should be generally programmable, and thus allow the use of arbitrary memory safety techniques.

Requirement 3 (Cross Layer)**.** Isolation mechanisms should provide programmers explicit access to microarchitectural state that impact data safety.

MicroStache implements all the stated requirements, being based on a simple hardware enforced isolation scheme, and exposing simple load/store memory primitives, thereby incurring minimal performance overhead on applications. Moreover, MicroStache provides a special cache for sensitive data, with an appropriate maintenance interface.

3 Threat Model and Assumptions

MicroStache prevents attacks on sensitive data integrity and confidentiality, coming from both malicious user inputs and microarchitectural side channels. We assume a program to be buggy but not malicious, and that an attacker can invoke arbitrary regular reads and writes. When combined with memory safety approaches (as described in Section 5), MicroStache defends against all control-flow hijack attacks. Implementing alternative safety solutions would lead to diverse threat models. We assume all hardware, operating system (OS), and compiler configuration to be correct, and read-only code and non-executable data.

We assume that the adversary can observe the victim's use of the processor cache, but that the adversary cannot observe the data values in the cache. We assume that the adversary has access to the source code of victim application, both before and after transformation using our compiler, and that the adversary cannot directly observe the victim's secret input data either due to encryption or due to isolation enforced by the OS. We do not address resource exhaustion attacks resulting from e.g. cache line locking abuse.

4 MicroStache Design

MicroStache supports the *safe region* paradigm by using the instruction elevation approach, where the safe region is placed in an external memory segment, the stache, and is only accessible through MicroStache instructions. The stache is an independently addressed segment of physical memory that uses offset-based addressing—bypassing virtual memory altogether. In this way, MicroStache integrates an isolated execution domain into the process without requiring domain switching. Figure 1 illustrates the main design elements of MicroStache. An overview of the MicroStache instruction set and their semantics are presented in Table 2. In the following we detail (i) the basic stache design, (ii) an extension that supports stack relative addressing and restricts stack access to the current frame locals (excluding return address and frame pointers), and (iii) an extension for mitigating cache side channel attacks.

4.1 Stache Segment

As depicted in Figure 1 the stache segment is a linear region of physical memory, and its location is dynamically specified by two new privileged registers: the stache base (`xbase`) and stache end (`xend`). The operating system virtualizes the stache by storing its base and bound as a part of thread state, giving each process a unique region of physical memory, and not mapping their contents into any address space. The stache is accessed through load/store operations, `xld` and `xst`, which takes the address as (`xbase` +offset). The hardware limits all access to the (`xbase`,`xend`) region (avoiding arbitrary access to any physical address). A *safe region* application can use the stache by allocating data to it, and with compiler support, emit stache access instructions for its secure access.

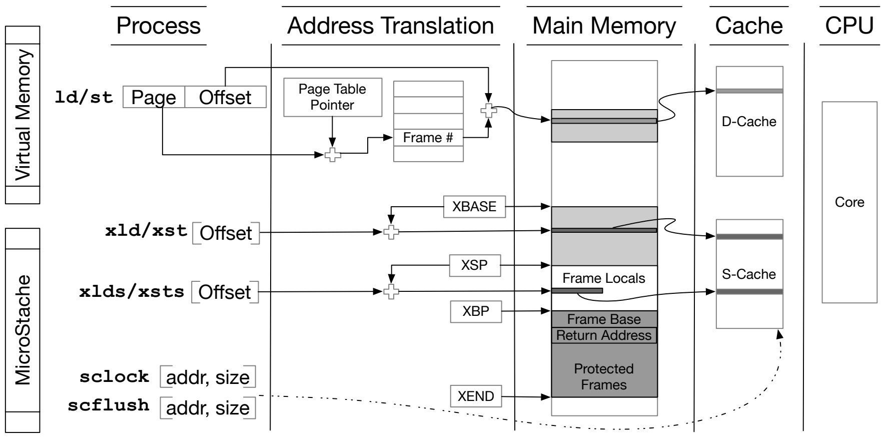

Fig. 1. MicroStache architecture.

Table 2. MicroStache interface. `reg` are general-purpose architecture registers. `addr` are memory addresses. `call` and `ret` also retain their native call/return semantics.

Abstraction	**Hardware Operation**	**Semantics**
Memory	`ld/st reg, addr`	Access regular mapped memory.
	`xld/xst xbase, addr`	Access stache segment, relative to `xbase`.
	`xlds/xsts xsp, offset`	Access Safe Stack memory, relative to `xsp`.
Safe Stack	`call addr`	Initialize new frame on Safe Stack.
	`xalloc size`	Allocate space on the Safe Stack.
	`ret`	Pop current frame from the Safe Stack.
S-cache	`scflush addr, size`	Flush S-cache lines given by `addr` and `size`.
	`sclock addr, size`	Lock S-cache lines given by `addr` and `size`.

4.2 Safe Stack

In order to efficiently isolate stack data, MicroStache includes stack hardware support through the following interface: frame creation (`call`), memory allocation (`xalloc`), frame destruction (`ret`) and stack-pointer-relative stache access (`xlds/xsts`) (see semantics in Table 2). The stack is located at the end of the stache region and grows towards lower addresses, and is only accessible through `xlds/xsts` instructions. Beyond standard stack operation, MicroStache guarantees that memory addresses computed for `xlds/xsts` are always in the range of `xsp` and `xbp` (the active frame) and that even stache relative access through `xld/xst` cannot modify the stack, thus ensuring that return address and base pointer corruption (which could be used for stack pivoting) is not possible. Moreover, the Safe Stack can be used to protect a subset of program control-data and decision-making non-control data, thus reducing the attack surface for Jump-

Oriented Programming (JOP) [5] and Data-Oriented Programming (DOP) [25] attacks. We quantify the value of this design in Section 7.

4.3 Safe Cache

A goal of MicroStache is to mitigate covert information channels without sacrificing application performance. For this purpose, MicroStache routes all sensitive memory accesses through a special cache, the safe cache (S-cache). The S-cache is a L1 cache similar to the data cache (D-cache), using the same microarchitecture-defined line update and eviction policy, but accessible only through MicroStache load and store operations. This is similar to other static cache partitioning schemes, such as Intel CAT [33].

This design point is sufficient to provide basic separation between regular and sensitive data, but it does not protect against side channel attacks on the S-cache. To make this possible, we add two new operations, `scflush` and `sclock`, that can be used by programmers to flush and lock cache lines respectively. `scflush` can be used to flush and invalidate cache lines, or the entire cache. Using `scflush`, applications and/or the operating system can implement simple policies such as flushing the S-cache on context switches. However, we expect this policy to have a significant negative impact on performance. Thus programs can prevent attackers from flushing or evicting their cache lines from the S-cache using `sclock`. However, we note that `sclock` must still be used correctly in order to ensure cache side channel protection.

5 Security Applications

In this section we describe how to use MicroStache for improving application security with minimum performance costs. In many cases, using MicroStache is a direct translation of safe region use: allocate safe-region data to the stache and issue MicroStache memory access for elevated access.

5.1 Safe Stack Protection

The safe stack approach [30] splits data on the stack into safe and unsafe based on the observation that a subset of locally-scoped data can be statically proven safe. Thus programs can make use of two separate stacks, the safe stack and the unsafe stack, to hold local variables. This technique can be automatically applied to existing programs with minimal overhead and no compatibility loss.

MicroStache protects the safe stack by modifying the existing SafeStack software infrastructure to emit MicroStache instructions instead of traditional stack instructions. More specifically, we want to (i) allocate data on the Safe Stack on function entry, using `xalloc`; and (ii) safely access local variables using `xlds`/`xsts`. This limits safe stack management to MicroStache primitives, ensuring strong Safe Stack protection against unintended memory accesses.

5.2 Code-Pointer Integrity

Code-Pointer Integrity (CPI) [30] is a technique that detects corruption of code pointers, eliminating all control-flow hijack attacks. In CPI, all pointers that may be used as a target for an indirect call or return operation are protected by placing pointer metadata and bounds information into a safe region. A compiler instruments all legitimate definitions of each pointer with a call into the CPI runtime to update pointer metadata. Then on each pointer use, CPI checks that the last update was legitimate. By using this Data-Flow Isolation (DFI) policy [9], CPI gains guaranteed control-flow hijack defense. MicroStache is used by modifying the CPI runtime to allocate metadata into the stache and replacing normal loads and stores with stache instructions.

5.3 Secret Pointer Protection

Modern systems make memory corruption attacks harder by relying on information hiding mechanisms such as ASLR [20,11]. What this effectively means is that pointers to locations of certain sections of the program, e.g. code, data, are hidden, and given to a few elevated instructions at runtime. Unfortunately, ASLR is susceptible to information leak attacks through memory corruption and cache and timing side channels [21]. We propose preventing secret pointer leaks by storing sensitive data in the stache segment. In our proof-of-concept work (Section 6) we show that hiding the location of the CPI safe region can be easily achieved by accessing the pointer through MicroStache, and that it requires minimal modifications to the CPI run-time.

5.4 Secret Computation Defense

We consider the general problem of information leaks through system-level attacks using cache side channels. As long as regular (non-sensitive) data doesn't depend on them, sensitive scalar values, e.g. integers, are trivially protected by MicroStache, because the computations performed, e.g. arithmetic operations, are invariant with respect to cache usage. Composite values, e.g. arrays, however may incur side channels depending on the computation that is being performed: if data dependencies involved in the computation lead to variations in cache access patterns, then these patterns can reveal parts of the data to an attacker with partial control over the cache.

We provide the possibility of efficiently performing secret computation through MicroStache by allowing programmers to lock small amounts of data into the S-cache. Listing 1.1 illustrates a simple scenario in which a global array, `secret`, is subject to computation that could cause leaks via cache side channels, e.g. data encryption. We make it impossible for attackers to infer bits of `secret` by locking it into the S-cache using the `sclock` primitive.

Note that this approach is limited to small data, i.e. not over the S-cache size. In this case, MicroStache can be combined with other hardware or software techniques, as discussed in Section 8.

```
int secret[ARRAY_SIZE]
  __attribute__((secret));
int f(int input)
{
  int result;
  sclock(secret, sizeof(secret));
  /* secret computation */ \ldots
  return result;
}
```

Listing 1.1. Computation on secret data using `sclock`.

6 Implementation

This section presents the implementation of our MicroStache prototype including: microarchitectural simulation in Gem5, LLVM compiler support, and security application details.

6.1 Gem5 Hardware Prototype

We built a proof-of-concept prototype of MicroStache for the x86 architecture. Our implementation consists of a simulated hardware prototype built using Gem5 [4]. We extended the x86 Gem5 model with MicroStache support: we added new registers, x86 micro-ops for memory operations and associated macro-ops by extending the decoder, execution and memory access Gem5 components; we extended `call` and `ret` x86 operations with Safe Stack support; finally, we extended the TimingSimpleCPU[4] generic CPU model with a new port for stache memory accesses. In a typical scenario, Gem5 MicroStache configuration involves connecting the memory port to the S-cache, which is then connected to the system interconnect. Additionally, we added support for MicroStache in the Gem5 system call emulation mode for stache initialization.

Our current MicroStache prototype does not support S-cache flushing, S-cache locking and automatic stack frame unwinding on `ret`. We emulated S-cache programmable operations by manually loading and/or replacing data in the S-cache. Similarly, we emulated automatic stack frame unwinding through a special stack deallocation instruction.

6.2 Software Support

Integrating MicroStache with existing software requires minimal software support in the compiler toolchain, i.e. assembler (emitting MicroStache opcodes) and C compiler (MicroStache intrinsics). To demonstrate MicroStache compiler support, we added opcode emission support to both LLVM 3.8 and 3.2. We implemented high-level support for the MicroStache instruction set in two ways: partial backend support for the LLVM instruction selection passes for

[4] `http://www.gem5.org/SimpleCPU`

alloca, load and store instructions; and a small run-time comprising general-purpose ustache_alloca, ustache_load and ustache_store functions for local and global variables respectively. We used the first implementation to automatically generate instrumented code for Safe Stack use, and the second implementation to implement CPI, secret pointer, and secret computation protections.

6.3 Security Applications

We implemented Safe Stack support by: modifying the existing SafeStack pass in LLVM 3.8; and adding backend support for MicroStache safe stack frame management and load/store accesses to the safe stack. We modified the SafeStack instrumentation to leave unsafe allocations on the regular stack, and move safe allocations to the MicroStache Safe Stack, by replacing safe allocas with xallocs. Then for load/store instructions to xalloc frames, we emitted xlds/xsts instructions in the target-dependent instruction selection phase. In our work we first attempted to extend LLVM with the MicroStache memory model, which we found was extremely challenging due to the complexity of modelling Safe Stack frames in the target-independent instruction selection passes. Although this is an engineering limitation from MicroStache's perspective, it is an open area to explore how to add non-standard memory models to LLVM's backend.

In order to create a more robust toolchain we switched to LLVM 3.2 which had both SafeStack and CPI passes implemented. However, in this case we only added general stache memory access support instead of just the Safe Stack. To protect the location of the CPI safe region for randomization protection, we modified the CPI run-time to load and store the pointer to the safe region at the appropriate times. More exactly, we modified cpi_init to store the pointer after the initial mmap. Similarly, we modified cpi_get and cpi_set to load the safe region pointer from the stache segment. Due to the inability of Gem5 to load dynamically linked executables, we did not protect the GOT. Finally, we extended the LLVM 3.2 CPI run-time with full MicroStache support, using xld and xst to manage metadata on the stache.

We implemented two popular [26,42] scenarios involving secret computation: Dijkstra's Single Source Shortest Path (SSSP) algorithm, and Top-k selection. The two algorithms operate on secret data structures: a graph and a binary heap. We used the Gem5 system call emulation (SE) mode to emulate S-cache locking by manually loading data into the S-cache before the actual computation.

7 Security and Performance Evaluation

We discuss three aspects pertaining to our MicroStache prototype: we evaluate the security of MicroStache using a safe stack as a reference scenario; we quantitatively and qualitatively analyze the security of each of the security applications presented in previous sections; and we measure the execution performance of our MicroStache prototype using standard benchmarks as well as the proof-of-concept scenarios presented in Section 6.

We ran all the experiments on the Gem5 TimingSimpleCPU model [4]. The TimingSimpleCPU model, in addition to instruction execution timing, simulates memory access latencies. We configured a basic Gem5 system comprising a CPU running at 1GHz, an L1 64 KB D-cache and 32KB I-cache, an S-cache, and a DDR3 memory controller running at 1600MHz. We connected all the caches to the memory controller through the default Gem5 system cross-bar memory bus.

7.1 Safe Stack Security Evaluation

We evaluate the effectiveness of our safe stack protection using a security benchmark suite similar to RIPE [45]. We implemented attack scenarios for ROP, JOP and DOP, using memory corruption vectors on the stack: return addresses, function pointers, `setjmp` buffers and local variables used for conditional branches. We compiled the tests and ran them using our MicroStache prototype. All the benchmarks pass, with the exception of `setjmp/longjmp`, because `setjmp` buffers aren't protected in our prototype. We observe otherwise that our MicroStache prototype trivially protects sensitive local variables such as function pointers.

7.2 Security Analysis

We analyze the security of the applications designed in Section 5. More specifically, we: measure the effectiveness of Safe Stack at reducing ROP [6], JOP [5] and DOP [25] attack surface; discuss the effectiveness of in protecting secret pointers; and qualitatively analyze the effectiveness of our protection mechanisms against cache side channel attacks.

To measure the attack surface reduction of our Safe Stack protection, we define a new static attack surface metric that tracks the number of *protected* branches, i.e. branches that are taken correctly as a result of their dependency on data placed on the Safe Stack. Thus, in our static analysis, a given control-flow transfer instruction is considered protected if it depends exclusively on safely accessed data. The Attack Surface Reduction (ASR) metric is:

$$\text{ASR} = \frac{\text{protected branches}}{\text{protected branches} + \text{unprotected branches}}$$

Safe Stack Return-Oriented Protection The basic MicroStache Safe Stack provides full safety for function returns, even without any explicit *safe region* use. This ensures that in the event of control-flow hijack attacks, the attacker cannot obtain control of the program using ROP gadgets.

Safe Stack Forward Indirect Branch Protection We determine the number of protected indirect forward flows at the LLVM IR level by computing the set of indirect call and indirect branch instructions. The results of this analysis are shown in Figure 2. Overall, we observe a significant number of indirect branches protected while only protecting safe local variables, indicating a powerful element to new hybridized mitigations for JOP attacks.

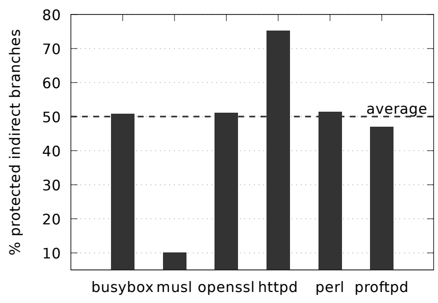

Fig. 2. ASR indirect forward flows.

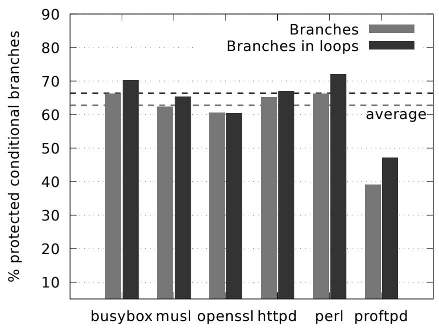

Fig. 3. ASR conditional branches.

Safe Stack Data-Oriented Protection We determine conditional branch protection similarly to indirect forward flows. The results are shown in Figure 3. On average, MicroStache protects approximately 62.7 % of static branches over all the analyzed programs. Additionally, we wish to assess protection against DOP [25]. DOP attacks involve controlling conditional branches to access data-oriented gadgets. However, in order for arbitrary execution to be possible, the gadgets must be controlled using a gadget dispatcher, i.e. branch instructions must reside in a loop. Thus we determine the reduction of unsafe branches in loops, which on average is approximately 66.3 %. In order to determine whether we eliminate any of the known DOP vulnerabilities, we manually analyzed the examples given by Hu et al. [25]. We found out that MicroStache protects all the branch condition variables in the ProFTPD example, which makes DOP impractical, if not impossible. We believe that a similar level of protection is provided against Control-Flow Bending (CFB) attacks [7].

Secret Pointer Protection MicroStache can be used to protect secret pointers, i.e. pointers that point to sensitive memory locations. We illustrate this by protecting the location of the CPI safe region (Section 6). This ensures that the safe region is accessed through a secure interface, which removes information leaks through memory corruption. Moreover, protecting the GOT and other secret pointers can further reduce the attack surface, eliminating part of the assumptions about the layout of the address space made by Evans et al. [17].

Secret Computation Defense MicroStache implicitly provides protection against cache side channels for scalar values when they are not part of branch conditions, since their cache location is invariant and they do not incur data dependencies measurable by the attacker. Protecting variables used in conditional branches is possible through branch normalization, either manually or by using compiler-based approaches such as Escort [37]. Composite values can be protected using `sclock`. In order to protect secret data for Single Source Shortest Paths (SSSP), we store and lock both the graph (our secret) and the computed distances. The reason for also storing the computed distances is that the computation of shortest paths depends on the structure of the graph, and thus cache access patterns would otherwise leak whether an edge exists between two nodes. Similarly, for Top-k selection we only store and lock the binary tree, since the only data dependencies occur between tree nodes. Thus in the worst case, the attacker can infer the size of the data being locked by inspecting S-cache access patterns.

7.3 Performance Evaluation

Our performance evaluation aims to determine: (i) the performance impact of our modifications on the Gem5 simulator; (ii) the performance impact of MicroStache enforced CPI and SafeStack relative to randomization based protection; (iii) the performance overhead of secret pointer protections; (iv) the performance overhead of cache side channel defense; and (v) the impact of S-cache size on performance, in particular of automated safe stack instrumentation.

```
// Regular stack          // MicroStache
1: sub $8, %rsp           1: xalloc $8
2: mov %rcx, (%rsp)       2: xst %rcx, $0
3: mov (%rsp), %rcx       3: xld %rcx, $0
4: loop 1b                4: loop 1b
```

Listing 1.2. `loop` hot path implementation. The two versions access the regular stack and the Safe Stack using regular and MicroStache memory instructions.

Table 3. Micro-benchmark run-time overhead. The baseline is unmodified Gem5 accessing the regular stack; `regular` and `stache` represent the stack that is accessed (regular, and Safe Stack respectively); `loop` and `recursive` are two microbenchmark implementations which access the stack in a loop and recursively respectively.

Benchmark \Scenario	`regular`	`stache`
`loop`	0.032 %	−0.065 %
`recursive`	4.168 %	0.379 %

Microbenchmarks We aim to assess the impact on run-time performance of the Gem5 CPU modifications introduced in Section 6. MicroStache memory access instructions and their regular counterparts should have similar run-times, as we use the same logic for memory access requests; the only difference is the memory ports used, i.e. the S-cache instead of the D-cache. We wrote two microbenchmarks that allocate space on the safe stack and read/write the allocated memory: the first, `loop`, uses the x86 loop instruction to do this iteratively; the second, `recursive`, does the same operation by calling a function recursively. We implemented two variants for each micro-benchmark, one that reads/writes to the regular stack using x86 `mov` instructions, and one using `xlds`/`xsts`. An example of the hot path in `loop` is given in Listing 1.2. We set the micro-benchmark to run the code in the loop a million times. The run-time performance of the two scenarios relative to the baseline is presented in Table 3. We observe that the performance is almost the same in each of the scenarios, with the exception of the `recursive` benchmark in the `regular` case, which has approximately 4 % overhead. There are two major causes for the large overhead: the modifications to `call` and `ret` (Section 6), which cause it to push the return address to both the stache segment and the regular stack; and the difference in instruction size and alignment between `regular` and `stache`. The latter influences the performance of the Gem5 TimingSimpleCPU fetch unit, which prefetches 8-byte words on x86-64. This behaviour is expected to occur on real x86 processors, and thus we assume the compiler optimizes for it in real applications.

Safe Stack SPEC benchmarks To validate our MicroStache prototype, we compiled and ran a small subset of the SPEC CPU2006 benchmarks using our modified SafeStack LLVM pass: `gobmk`, `hmmer`, `lbm` and `specrand`. Figure 4 shows the performance of `safestack` using randomization based protection and `stache` relative to the baseline. We believe that the reasons for the high overhead is similar to that observed in the microbenchmarks, and can be further optimized at the compiler level. Many of the test did not compile due to the challenges of extending LLVM with the non-standard MicroStache memory model.

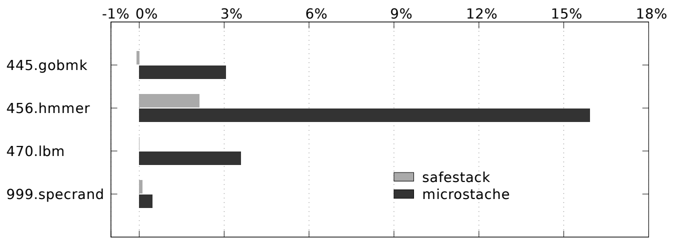

Fig. 4. SPEC CPU2006 run-time performance overhead for safe stack relative to mainline Gem5 and LLVM 3.8; `safestack` represents mainline Gem5 and LLVM 3.8 SafeStack; `microstache` represents MicroStache Gem5 and MicroStache LLVM 3.8.

Secret Pointer Protection Test To measure the performance impact of hiding secret pointers, we ran a small test suite for our modified version of the CPI run-time library against the original. The results show less than 1 % overhead, further demonstrating of the minimal impact of MicroStache hardware.

Code-Pointer Integrity SPEC Benchmarks We instrumented the SPEC benchmarks using the ASLR based implementation of SafeStack (*i.e.,* using the normal stack and regular ld/st instructions) and three variants of CPI: ASLR, SFI and MicroStache. Figures 5 illustrates the results relative to native Gem5 CPU baseline. The goal of this benchmark is to demonstrate how MicroStache performs with respect to the randomized version (no checks while using normal ld/st instructions) and the more costly SFI variant of CPI.

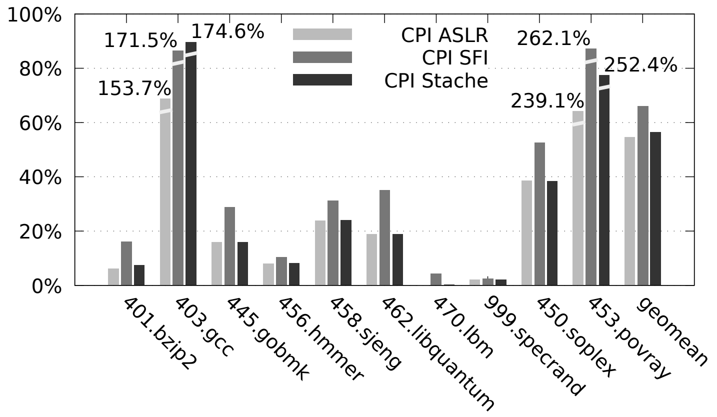

Fig. 5. CPI overhead relative to native on SPEC. On average MicroStache is 1.2% slower than randomized and 7.4% faster than SFI.

Cache Side Channel Benchmarks Table 4 shows the performance overhead of the cache side channel protection. We compare the performance of MicroStache with Escort [37], a software solution. We observed that MicroStache is comparable with the baseline performance, and in particular in one of the two scenarios, outperforms the baseline by about 26 %. The Escort prototype has a high performance overhead in the `sssp` benchmark because it relies on Intel AVX for fast memory updates, a feature which is not available in our Gem5 MicroStache prototype. The data in our MicroStache benchmark implementation is small enough that it fits in the S-cache. This would be problematic for large secret data. However, since MicroStache and Escort use orthogonal mechanisms for computation involving secrets, in principle they can be used together in the case when the secret data is too large to fit into the S-cache.

S-cache Size Impact on Performance Adding the S-cache to the system constitutes a trade-off between processor die space and stache segment access latency.

Table 4. Cache side channel protection runtime overhead. Escort and MicroStache are measured relative to the baseline (unprotected) benchmark versions. `sssp` is a Dijkstra single-source shortest path (SSSP) implementation; `top-k` is a Top-k selection implementation.

Benchmark \System	Escort	MicroStache
`sssp`	87.19 %	−26.02 %
`top-k`	0 %	2.05 %

Table 5. SPEC CPU2006 maximum run-time stache size.

Benchmark	stache size (B)
445.gobmk	440
456.hmmer	808
470.lbm	96
999.specrand	72

Thus we measured the maximum run-time safe stack size, results are shown in Table 5.

8 Discussion and Future Work

In this section we discuss MicroStache limitations and outline approaches to overcome them; and we compare MicroStache with closely related work, namely HDFI [41]. **Cache Side Channel Analysis:** To provide complete cache side channel protection using MicroStache, we plan to leverage automated techniques for side channel analysis [36,32,37]. Further, we aim to extend our evaluation to comprise an empirical analysis of cache side channel protection, by simulating a full-system scenario using Gem5. **Exceptional Control Flows:** While our safe stack MicroStache prototype (Section 6) can be used as a shadow stack, it does not provide support for exceptional control flows, such as `setjmp/longjmp` and `try/catch`. We plan to support this by integrating with related work [14,30,10]. **Comparison With HDFI:** Song et al. [41] propose fine-grained memory isolation through a hardware element called hardware-assisted data-flow isolation (HDFI). While HDFI and MicroStache have similar goals, HDFI achieves them through static 1-bit tagging, while MicroStache uses a statically-allocated memory region. Both options have benefits. HDFI tagging leaves data in place but requires a tag cache and lookup operations which impacts performance and hardware complexity, whereas MicroStache is simpler, and it can be used for a larger set of applications such as general memory safety techniques.

9 Conclusion

In this paper we explored the abstractions necessary for efficient in-address space *safe region* protection. We proposed MicroStache, a new microarchitectural isolation mechanism designed on the principle that safely accessed data can be efficiently protected by separating memory accesses at multiple abstraction levels (ISA, cache, main memory). We showed that the programmability of MicroStache allows it to be employed in a variety of use cases, with no to minimal overhead. In combination with existing compiler techniques, MicroStache can be leveraged to efficiently protect a large subset of local variables and sensitive control-flow transfers, significantly reducing the surface of jump-oriented and data-oriented attacks, as well as information leaks and memory corruption. Thus we believe MicroStache to be a significant element for fine-grained, flexible and efficient sensitive data isolation.

10 Acknowledgments

The authors would like to thank the anonymous reviewers, Mohit Tiwari, André DeHon, JMS, Volodymyr Kuznetsov, and George Candea, who all provided insightful feedback, improving the paper. We would like to thank our families—Filip, Andi, Bogdan, Andreea, Cornelia, Vasile, and AuNoLeZeZo—who gave us the freedom and flexibility to make this paper the best it could be. This research was funded in part by National Science Foundation grant CNS-1513687. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, et al. SCONE: Secure Linux containers with Intel SGX. 2016.
2. A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.
3. A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.
4. N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.
5. T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.
6. E. Buchanan, R. Roemer, S. Savage, and H. Shacham. Return-oriented programming: Exploitation without code injection. *Black Hat*, 8, 2008.
7. N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, 2015.
8. S. A. Carr and M. Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.
9. M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160. USENIX Association, 2006.
10. N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis. HCFI: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 2016.
11. S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE Symposium on Security and Privacy*, 2015.
12. J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *ACM SIGOPS Operating Systems Review*, 2007.
13. N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. *ACM SIGPLAN Notices*, 50(4):191–206, 2015.
14. L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM*

Symposium on Information, Computer and Communications Security, pages 40–51. ACM, 2011.

15. J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. In *ACM SIGARCH Computer Architecture News*, 2008.
16. U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. *SIGARCH Comput. Archit. News*, 43(1):487502, Mar. 2015.
17. I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy*, 2015.
18. D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.
19. T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi. IMIX: In-process memory isolation extension. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.
20. C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained Address Space Randomization. In *USENIX Security Symposium*, pages 475–490, 2012.
21. B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Guiffrida. ASLR on the line: Practical cache attacks on the MMU. In *Network and Distributed System Security Symposium*, 2017.
22. D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.
23. L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *IEEE Symposium on Security and Privacy*, 2015.
24. L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. TrustShadow: Secure execution of unmodified applications with ARM TrustZone. *arXiv preprint arXiv:1704.05600*, 2017.
25. H. Hu, S. Shinde, A. Sendroiu, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy*, 2016.
26. I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.
27. T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, 2012. USENIX.
28. K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.
29. D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

30. V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.
31. W. Li, Y. Xia, H. Chen, B. Zang, and H. Guan. Reducing world switches in virtualized environment with flexible cross-world calls. In *ACM/IEEE 42nd Annual International Symposium on Computer Architecture*, 2015.
32. C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGARCH Computer Architecture News*, 43(1):87–101, 2015.
33. F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE International Symposium on High Performance Computer Architecture*, 2016.
34. S. Nagarakatte, M. M. Martin, and S. Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.
35. S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):245–258, 2009.
36. A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium*, 2015.
37. A. Rane, C. Lin, and M. Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *USENIX Security Symposium*, 2016.
38. N. Roessler and A. DeHon. Protecting the stack with metadata policies and tagged hardware. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 1072–1089.
39. D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security Symposium*, 2010.
40. M. S. Simpson and R. K. Barua. Memsafe: ensuring the spatial and temporal memory safety of c at runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.
41. C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. HDFI: hardware-assisted data-flow isolation. In *IEEE Symposium on Security and Privacy*, 2016.
42. S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 2010.
43. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, 1994.
44. Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.
45. J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.
46. B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy*, 2009.