# Rapid Prototyping for Microarchitectural Attacks

Catherine Easdon
*Dynatrace Research*
*Graz University of Technology*

Michael Schwarz
*CISPA Helmholtz Center for Information Security*

Martin Schwarzl
*Graz University of Technology*

Daniel Gruss
*Graz University of Technology*

## Abstract

In recent years, microarchitectural attacks have been demonstrated to be a powerful attack class. However, as our empirical analysis shows, there are numerous implementation challenges that hinder discovery and subsequent mitigation of these vulnerabilities. In this paper, we examine the attack development process, the features and usability of existing tools, and the real-world challenges faced by practitioners. We propose a novel approach to microarchitectural attack development, based on *rapid prototyping*, and present two open-source software frameworks, *libtea* and *SCFirefox*, that improve upon state-of-the-art tooling to facilitate rapid prototyping of attacks.

*libtea* demonstrates that native code attacks can be abstracted sufficiently to permit cross-platform implementations while retaining fine-grained control of microarchitectural behavior. We evaluate its effectiveness by developing proof-of-concept Foreshadow and LVI attacks. Our LVI prototype runs on x86-64 and ARMv8-A, and is the first public demonstration of LVI on ARM. *SCFirefox* is the first tool for browser-based microarchitectural attack development, providing the functionality of *libtea* in JavaScript. This functionality can then be used to iteratively port a prototype to unmodified browsers. We demonstrate this process by prototyping the first browser-based ZombieLoad attack and deriving a vanilla JavaScript and WebAssembly PoC running in an unmodified recent version of Firefox. We discuss how *libtea* and *SCFirefox* contribute to the security landscape by providing attack researchers and defenders with frameworks to prototype attacks and assess their feasibility.

## 1 Introduction

Since seminal early work on hardware timing channels in the 1990s [43, 59, 133], software-based microarchitectural attacks have significantly impacted the modern security landscape. Such attacks can be categorized by the data security property they compromise: confidentiality, integrity, or availability.

Microarchitectural side-channel attacks compromise data confidentiality, leaking metadata (such as memory access patterns) to infer the actual data from. These attacks were historically used to attack cryptographic primitives [3, 9, 85], but more recently have also been applied to other targets such as inter-keystroke timing [41, 64, 80, 93, 126]. A collaborating victim and attacker (*i.e.*, a sender and receiver) can use these same techniques to establish a microarchitectural covert channel to exfiltrate data from a system [87]. This recent attack paradigm is particularly powerful as it enables data to be directly leaked from internal CPU buffers [14, 47, 49, 50, 90, 100, 106, 114, 115, 123].

In contrast, microarchitectural fault attacks, which induce faults via rapid memory accesses [39, 53, 56, 61, 63, 103, 120], undervolting [55, 81, 89] or clock manipulation [109], compromise data integrity by exploiting the physical properties of hardware from software. Finally, microarchitectural denial of service attacks compromise data availability, for example by degrading performance [7, 124] or halting the system [46, 54].

A common characteristic of all these attacks is that they are complex to implement [8, 35, 137]. Contributing factors include the complexity of modern microarchitectures, lack of documentation of microarchitectural features, public unavailability of microarchitectural debugging and tracing tools, and noise induced by undocumented microarchitectural behavior or system activity. Hence, successfully implementing a microarchitectural attack can be extremely challenging, even with a strong conceptual understanding of how the attack works. These challenges hinder attack discovery and mitigation in both academia and industry, as we explore in Section 3.

One approach to tackling the complexity of attack development is automation. Differential analysis [131, 134] and templating [35, 41] can be used against applications to generate new attack variants from known attack types. For example, ABSynthe [35] treats the CPU as a black box and generates leakage maps for different instruction pairs to synthesize contention-based covert channels. Osiris [130] tests all CPU instructions for interferences to find new timing-based covert channels, while Transynther [79] uses mutation-based fuzzing to synthesize new Meltdown variants from existing attack implementations.

However, these automated techniques are typically limited to finding new variants of known attacks, rather than entirely new attack types. Many of today's known microarchitectural attacks introduced entirely novel techniques [55, 56, 58, 67, 81, 88, 117, 118], a new threat model [63, 97, 102, 111, 114], or an entirely different perspective, e.g., switching the attacker and victim to turn existing attacks around [58, 61, 115]. Finding new attacks is therefore often still the result of manual experimentation and analysis. Thus, in this paper we seek to answer the question: *How can we reduce the software implementation burden for **manual** development and reproduction of microarchitectural attacks?*

Informed by an investigation of the microarchitectural attack development process, we develop two frameworks. The first framework, *libtea*, provides an API that abstracts away platform-specific implementation details such as native timers, enabling rapid development of cross-platform attack prototypes. The second framework, *SCFirefox*, exposes this API to JavaScript in a modified Firefox browser to help researchers assess the viability of browser-based microarchitectural attack variants. Access to high-resolution timers, address information, and the ability to modify page-table entries from JavaScript enables rapid prototyping of browser-based attacks. Once their feasibility is determined, these prototypes can be gradually ported to vanilla JavaScript and WebAssembly (WASM).

While this work does not itself introduce a novel attack type, we hope that our frameworks and rapid prototyping methodology will establish a strong foundation for practitioners to do so in the future. Additionally, we contribute to improved understanding of microarchitectural attacks, firstly by demonstrating the first Load Value Injection (LVI) [115] attack on ARMv8-A, and secondly by prototyping a browser-based ZombieLoad [100] proof-of-concept (PoC) to present the first Microarchitectural Data Sampling PoC for an unmodified browser. We show that cache-line flushing is *not* necessary to induce ZombieLoad leakage via microcode-assisted page-table walks (variant 3), extending the scope of ZombieLoad attacks to constrained environments such as JavaScript sandboxes.

**Contributions.** The contributions of this work are:

1. We investigate the microarchitectural attack development process and evaluate existing attack frameworks.
2. We introduce *libtea* and *SCFirefox*, two open-source cross-architecture and cross-platform frameworks for rapid prototyping of attacks in native code and in JavaScript.
3. We prototype a Foreshadow PoC, the first Load Value Injection (LVI) attack on ARM and the first browser-based ZombieLoad attack.
4. We directly port our *SCFirefox* ZombieLoad prototype to vanilla JavaScript and WASM to create the first Microarchitectural Data Sampling attack for an *unmodified* browser, achieving a leakage rate of 1.48 B/s in Firefox 81 by exploiting page deduplication.

**Outline.** In Section 2 we provide relevant background. Section 3 presents our analysis of the attack development process. Section 4 presents our frameworks, and Section 5 demonstrates their utility by developing a Foreshadow PoC, the first LVI attack on ARMv8-A and the first browser-based ZombieLoad attack. We discuss avenues for future work in Section 6 and conclude in Section 7.

## 2 Background

In this section, we introduce the relevant technical background required for understanding the rest of the paper.

### 2.1 CPU Microarchitecture

The *microarchitecture* of a CPU refers to its specific hardware implementation of the abstract *architecture* defined by the CPU's instruction set architecture (ISA). Typically, events in the microarchitecture are transparent to the programmer and its state cannot be directly queried. Microarchitectural concepts of particular relevance to this work are the memory hierarchy, out-of-order execution, and speculative execution.

**The Memory Hierarchy.** Historically, memory access latency has been a bottleneck for execution on the CPU. To mitigate this, typical computing systems have a hierarchical memory structure, using progressively smaller and faster memory sections closer to the CPU to exploit temporal and spatial locality of memory accesses. CPUs typically employ one or more levels of fast core-private caches, e.g., L1, and a slower last-level cache (LLC) shared between cores, e.g., L3 on most x86 cores and L2 on many Arm SoCs.

**Speculative and Out-of-order Execution.** These two performance optimizations are common in modern CPUs. With out-of-order execution, the CPU does not have to execute instructions in program order. Instead, instructions later in the instruction stream can be executed first if they do not depend upon earlier instructions. This technique improves overall performance, as instructions can be executed in parallel and high-latency instructions need not block the pipeline.

Speculative execution optimizes handling of conditional control-flow changes, *i.e.*, conditional and indirect branches. By predicting and speculatively executing the code path following a branch, the pipeline need not necessarily stall if the value of the branch condition is not yet known. In the case of a mispredicted branch, the speculatively executed instructions are discarded and architectural state changes are rolled back. However, microarchitectural state is not restored.

### 2.2 Microarchitectural Side-Channel Attacks

Microarchitectural side-channel attacks exploit side channels in the implementation of the CPU and memory hierarchy.

These attacks can be classified into cache-, memory-, and predictor-based side-channel attacks, categories which we explore in more detail below. Transient-execution attacks build upon these side channels to directly leak data rather than metadata.

**Cache and Memory Attacks.** Cache attacks exploit the timing differences between memory accesses that can be served by the caches and memory accesses that have to retrieve data from main memory. By design, if an address is cached (a *cache hit*) a load will be significantly faster than if it must be fetched from a slower location, e.g., the DRAM (a *cache miss*). Cache attacks can be divided into three main types. With Evict+Time [84], an attacker fills a part of the cache (e.g., a cache set) and measures how that influences the execution time of the victim. With Prime+Probe [84], an attacker fills a part of the cache and monitors whether any part is evicted when the victim is executed. With Flush+Reload [138] and variants such as Evict+Reload [41], an attacker directly monitors cache-state changes on memory that is shared with the victim, by actively flushing the line from the cache and measuring if it is reloaded when the victim is executed. Cache attacks have been demonstrated on data caches [41, 68, 84, 138], instruction caches [2, 4, 94], address-translation caches [36, 95], and DRAM buffers [88, 129]. Related attacks also attack the paging structures. These attacks usually rely on a threat model where the attacker controls the operating system [119, 129, 136], e.g., in the case of trusted execution environments (TEEs) such as Intel SGX.

**Predictor Attacks.** Predictors used by modern CPUs can also be exploited to leak side-channel information. Exploited prediction types include branch prediction [5, 22, 58], store-to-load forwarding [53, 95, 107], cache way prediction [66] and prefetching [104].

**Transient Execution Attacks.** Transient instructions are instructions that are executed but never retire, *i.e.*, they never become architecturally visible but change the microarchitectural state. The cause of transient instructions can be either misspeculation or out-of-order execution after an assist, e.g., due to an exception occurring. Transient-execution attacks exploit the fact that transient instructions have access to data that is architecturally inaccessible. Transiently-accessed data can be transmitted to the architectural domain via a microarchitectural covert channel. Based on the cause of transient execution, such attacks are categorized into Spectre-type attacks [15, 31, 57, 58, 60, 70] (mispredicted speculation) and Meltdown-type attacks [14, 15, 67, 100, 106, 114, 123] (lazy exception handling).

The original Meltdown attack [67] enables unprivileged attackers to read kernel data (*i.e.*, addresses with the User/Supervisor bit set) that is cached in the L1 data cache. Subsequent attacks have demonstrated that many other Meltdown variants exist. Foreshadow [114], for example, enables attackers to read unmapped pages (*i.e.*, addresses with the 'present' bit cleared), again provided that the data is cached in L1. Two notable subsets of Meltdown-type attacks are Microarchitectural Data Sampling (MDS) and Load Value Injection (LVI). The MDS attacks RIDL [123], ZombieLoad [100], and Fallout [14] exploit Meltdown effects to sample memory accesses as they pass through other microarchitectural buffers, such as the line-fill buffers and store buffer. Load Value Injection (LVI) [115] reverses the Meltdown effect, switching the roles of the victim and attacker processes in a manner akin to a Spectre attack. While Spectre injects attacker-controlled control flow, LVI injects attacker-controlled data into the victim's transient execution by triggering a fault in the victim.

### 2.3 Rapid Prototyping

In manufacturing, rapid prototyping describes a class of methodologies where physical prototypes of a product are created in order to quickly and iteratively improve upon a design. The estimated time and cost savings from the resulting increased productivity are 50-90% [16]. Rapid prototyping has also been successfully applied for decades in software engineering [34, 69], and in both domains prototypes play a variety of important roles in the product development process. They are invaluable not only for testing but also for experimentation and learning, evaluation of requirements, and communication of product ideas (e.g., across diverse teams, to management, or to clients) [16, 69]. Throughout this work we interpret the term 'rapid prototyping' as it is typically used in software engineering, *i.e.*, we do not refer to fully automated fabrication of prototypes, but rather to facilitation of iterative manual development.

## 3 The Microarchitectural Attack Development Process

In this section, we analyze the microarchitectural attack development process to determine which software implementation challenges practitioners face and how these might be facilitated. We observe that most practitioners informally follow an iterative process akin to rapid prototyping, and propose that prototyping could be accelerated via the introduction of two new software frameworks to improve upon state-of-the-art tooling.

### 3.1 Motivation

While microarchitectural attacks have been systematized [10, 15, 27, 74, 96, 108, 135], the *development process* for these attacks has, to our knowledge, never been examined. Investigating this process enables us to identify and facilitate the software implementation challenges involved. By doing so, we hope to accelerate and reduce the cost of microarchitectural attack discovery and mitigation. This cost can be extremely high, as was starkly illustrated by Meltdown and Spectre, attacks that built upon results from years of previous research [24]. Microsoft "mobilized hundreds of people" to respond [25], while at Red Hat mitigations involved more than 60 engineers and cost over 10 000 hours of engineering [73].

We consider the attack development process in three contexts: attack research, attack mitigation, and education. Reproducing an attack is crucial for attack mitigation to determine which systems are vulnerable and whether a mitigation is effective [33]. An attack may need to be reproduced from scratch: two of our industry interviewees reported that the affected parties often do not provide PoCs to organizations involved in the coordinated disclosure process, and there are also published attacks for which no public PoCs exist [14, 46, 48]. In the context of education, facilitating attack implementation lowers the barrier to entry for learning about microarchitectural attacks. While these attacks exploit hardware, they destroy security boundaries at higher software abstraction layers, e.g., process isolation. Microarchitectural security education is therefore crucial so that technology professionals are aware of the leaky "tower of abstractions" [74] they are building systems upon, regardless of which abstraction layer they work at.

Concretely, in order to reduce the software implementation challenges involved, we seek to answer the following research questions (RQs):

- **RQ1: Attack Building Blocks.** What are the fundamental software building blocks used?
- **RQ2: Microarchitectural Control.** How can fine-grained control over the microarchitecture be achieved?
- **RQ3: Languages and Tooling.** Which programming languages are used to implement attack building blocks? How usable are existing attack development tools?
- **RQ4: Process and the Role of Prototyping.** Are there process commonalities between different attacks, and between different practitioners? Are these sufficient for it to be meaningful to discuss an overall 'microarchitectural attack development process', and if so, what role does prototyping play?

## 3.2 Methodology

We performed a mixed-methods analysis, combining a review of prior academic work, data from 10 expert interviews, and a user study with 28 student participants.

**Literature Review.** We considered all papers published at the top four systems security venues CCS, NDSS, S&P, and USENIX Security between 2015 and 2020 (inclusive) that featured implementation of software-based microarchitectural attacks targeting the CPU ($N = 102$). The list of papers included is provided in the extended bibliography [28]. We also considered any published source code for each paper. Papers requiring external hardware or hardware modification were excluded, as were papers targeting physical side channels.

**Expert Interviews.** We interviewed 10 microarchitectural security professionals (five in academia and five in industry). The eligibility requirement was research or mitigation experience of at least two microarchitectural attacks. Interviewees were recruited via email, Twitter, and an online conference announcement. Of our academic interviewees, two were assistant professors and three were PhD students. Collectively, they had experience working on 37 microarchitectural attack papers in 9 different research groups across 6 countries. Our industry interviewees each had substantial recent experience of attack mitigation in industry. In total, this experience spanned 6 companies in two countries. Additionally, all five had published research in this area. All interviewees were informed of the research aims, explicitly gave their consent for their responses to be published anonymously, and were able to review this paper before submission.

**User Study.** We conducted a double-blind within-subjects user study in which we evaluated cache attack libraries as part of an introductory graduate course on side-channel security. We tested *Mastik* [137] and *cacheutils* [13] together with *libsc*, an early prototype of *libtea*. Over 6 weeks, 19 pairs of students implemented a cache covert channel, a cache template attack on keystrokes [41], and a KASLR break (via prefetch timing [38], or Data Bounce [14]). Each pair tested one library per task in a randomized order. We focused on evaluating the usability of these tools and their attack building blocks (RQ3). 28/38 students chose to participate in our survey. We provide demographic statistics in [28]. While our institution has no ethical review board, we carefully designed the study to address ethical concerns appropriately: the survey was opt-in and entirely independent from grading, and all study responses were collected anonymously. The libraries were anonymized for the students, with the library name replaced in the code and documentation (e.g., with 'library1') and other identifying information removed. None of the participating students had an academic relationship to our frameworks or the other libraries. To avoid bias, the authors did not teach the lab sessions, and conducted initial analysis of the results with the anonymized library names remapped by the course practitioners before finally analyzing student comments that deanonymized the frameworks, e.g., comments mentioning functionality specific to one framework.

## 3.3 RQ1: Attack Building Blocks

Rather than focusing on identifying the abstract primitives required for a class of attacks as in prior work (e.g., a disclosure gadget [25, 135]), we focused in our literature survey on identifying the concrete software constructs used to implement or debug these primitives. The following building blocks are most commonly used in prior work. (Note that [28] provides full citation lists for each block.)

1. High-precision timers (95/102) [6, 35, 101]

2. Cache-related building blocks (79/102), such as flush-based side and covert channels [7, 40, 138], eviction set generation and eviction-based side and covert channels [19, 52, 68], and knowledge of physical addresses, cache sets, or cache slices [36, 66, 127].
3. Transient-execution building blocks (43/102), such as branch predictor mistraining [15, 22, 58], exception handling [14, 67, 123], and memory and speculation fences [53, 79, 123].
4. Privileged building blocks (46/102), such as manipulation of page tables or memory protection attributes [14, 100, 123], manipulation of model-specific or system registers [55, 81, 116], execution in ring 0 [14, 15, 67], fine-grained execution control of TEEs [78, 94, 118], and custom interrupt handlers [62, 119, 129].

Similarly, the most useful building blocks for our study participants were Flush+Reload primitives including threshold calibration (16/28) and high-level primitives for timing, memory accesses, and cache line flushing (15/28). While these building blocks can be implemented in user-space, privileged building blocks such as page table modification are more challenging to implement. Examples of the required work include developing custom drivers [44, 119, 129], patching the Linux kernel [62, 77, 129], and even implementing a custom operating system (OS) or hypervisor [23]. However, most modifications are either not published [23, 44, 62, 129] or, in the case of kernel patches, are only available for kernel versions that are already end-of-life [77].

## 3.4 RQ2: Microarchitectural Control

McIlroy et al. [74] describe the challenges of modeling microarchitectural state and determining how programs can manipulate it as "massive open problems", as also confirmed by our interviewees. One approach is to rely on a custom operating system to achieve a higher degree of microarchitectural control. Microsoft use a custom OS for this purpose [23], and 7/10 of our interviewees also reported that they or their colleagues had previously used a custom kernel or hypervisor, most commonly for page-table modification (4/10).

An alternative approach is to use custom drivers or kernel patches with an existing mainstream kernel such as Linux. Google uses this approach internally with their SafeSide project [33] for end-to-end testing of software side-channel mitigations [92], as did 5/10 of our interviewees. Advantages include the reduced time investment required and the existing rich software ecosystem and hardware support of mainstream OSes. Moreover, our interviewees' frustration about noise most commonly concerned microarchitectural noise, such as hardware prefetching and the influence of attack code itself on microarchitectural state, rather than kernel-induced noise.

## 3.5 RQ3: Languages and Tooling

**Languages.** The majority of works are implemented in C/C++ and/or assembly (96/102). All of our interviewees also reported working predominantly in C/C++, using it as a wrapper around core attack code in assembly (9/10) or (for attacks also exploiting the GPU) graphics APIs such as OpenGL (1/10). A frequent comment was that it is infeasible to implement the core attack primitive in C/C++ as the resulting machine code may be different than intended (7/10).

While most interviewees used assembly liberally (8/10), many of our user study participants found higher-level wrappers around assembly primitives to be one of the most useful library features (15/28). This is an important consideration for attack tooling, as even experienced systems programmers may be unfamiliar with the particular assembly sequences used in microarchitectural attacks. (27/28 students reported either some or substantial experience in low-level C/C++ and operating systems, with 6 reporting substantial experience in both.)

The second most commonly used language in our review is JavaScript (18/102), either in pure browser-based attacks [36, 39, 101], or as a complement to PoCs in C [14, 58, 123]. WASM has the additional advantages of a linear memory and close proximity to machine code [70, 123, 127], while WebGL enables browser-based GPU attacks [26, 82].

**Existing Tooling.** We found 6 existing software frameworks for microarchitectural attack development.[1] *cacheutils* [13, 38, 41] is a single C header providing building blocks for Flush+Reload, Flush+Flush, and prefetch attacks on x86 and ARMv8-A. *Mastik* [137] supports trace-based cache attacks (Flush+Reload, Flush+Flush, Prime+Probe) and performance degradation attacks on x86. *libflush* [45, 65] provides support for Prime+Probe, Flush+Reload, Evict+Reload, Flush+Flush and prefetch attacks on x86, ARMv7, and ARMv8. *XLATE* [122] provides implementations of a range of cache attacks and cache-based covert channels for x86, notably featuring indirect cache attack primitives that exploit the MMU. *PTEditor* [98] enables user-space manipulation of page tables and memory protection attributes on x86 and ARMv8. *SGX-Step* [117] supports user-space page table manipulation, registration of custom interrupt handlers and call gates (enabling arbitrary C functions defined in a user-mode program to be run in kernel mode, as used by [81]), and local APIC timer configuration for single- and zero-step attacks against Intel SGX.

[1]We considered only tools intended for manual attack implementation, excluding complementary tools designed for automated attack discovery [35, 79], microarchitectural instrumentation, reverse-engineering, or profiling [1, 71, 110, 125], side-channel analysis of binaries [131, 132, 134], solely trace-driven analysis [94], and reproducible evaluation of existing attacks [76]. SpeechMiner [135] was excluded from analysis because it was not yet publicly available; we contacted the authors at the time to confirm this. CacheZoom [77] was excluded because it is unsupported on recent kernels.

| **Feature** | **Mastik** | **cacheutils** | **libflush** | **libsc** | **PTEditor** | **SGX-Step** | **XLATE** | **This work** |
|---|---|---|---|---|---|---|---|---|
| High-precision timers | ◐ | ◐ | ● | ● | ○ | ◐ | ◐ | ● |
| Flush-based cache attacks | ● | ● | ● | ● | ○ | ● | ● | ● |
| Eviction-based cache attacks | ● | ○ | ● | ● | ○ | ○ | ● | ● |
| Branch prediction mistraining | ○ | ● | ○ | ○ | ○ | ○ | ○ | ● |
| Exception handling or suppression | ○ | ● | ○ | ● | ○ | ◐ | ◐ | ● |
| Speculation fences | ● | ● | ○ | ● | ● | ● | ○ | ● |
| Obtain virtual/physical addresses and cache sets/slices | ◐ | ◐ | ◐ | ◐ | ◐ | ◐ | ◐ | ● |
| Read/write page tables and memory attributes | ○ | ○ | ○ | ○ | ● | ◐ | ○ | ● |
| Read/write model-specific/system registers | ○ | ○ | ○ | ● | ○ | ● | ○ | ● |
| Custom interrupt handlers and call gates | ○ | ○ | ○ | ○ | ○ | ● | ○ | ● |
| Fine-grained (TEE) execution control | ○ | ○ | ○ | ○ | ○ | ● | ○ | ● |
| Facilitates porting to the browser | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| Architecture(s) | x86 | x86, ARMv8-A | x86, ARMv7/8-A | x86 | x86, ARMv8-A | x86 | x86, ARMv7/8-A | x86, ARMv8-A |
| Operating systems (**L**inux, **W**indows 10) | L | L, W* | L | L | L, W | L | L | L, W* |

Table 1: A feature comparison of existing software libraries for microarchitectural attack development and the features of the *libtea* and *SCFirefox* frameworks. ●, ◐, and ○ indicate respectively full support, partial support, and absence of a feature. We use full and partial support to distinguish tools that are more fully-featured in a particular category, e.g., one tool may offer multiple timers while the other tools (partial support) provide only `rdtsc`. * indicates that not all features are supported on this OS.

Collectively, these frameworks have been used in 40 works (see [28]), demonstrating the utility of such tooling. Table 1 provides an overview of their features, with a focus on the attack building blocks identified in RQ1, and compares them to the functionality provided by our two frameworks. For comparison, we include *libsc*, an early prototype of *libtea* we evaluated in our user study to determine how to improve the API. This prototype was inspired by *cacheutils* and *libflush*, providing similar functionality for Linux x86 with additional utility functions for cache covert channels and support for reading model-specific registers.

As shown in Table 2, across all libraries and tasks the majority of our 28 user study participants agreed that the library made it faster to implement attacks compared to developing from scratch. For *cacheutils* and *libsc*, the majority agreed they reduced the amount of debugging required. Each tool had its strengths and weaknesses: *Mastik* was praised for its support for trace-based attacks (3/28) and consistent API for all attacks (3/28) but was difficult to build (17/28). Students appreciated the simplicity of *cacheutils* (16/28) and the fact it is entirely self-contained in one header (3/28) but found its low-level API and lack of documentation confusing (12/28). In contrast, the API of *libsc* was much preferred, as it combined more explanatory function names (e.g., `libsc_timestamp` versus `rdtsc`) (8/28) with better documentation (12/28) and configurable timers (3/28). However, for all libraries students requested more documentation and examples.

Of our interviewees, 3/10 had found *cacheutils* useful, while one had also used *Mastik* and *SGX-Step*. Generally, interviewees reported frequent code reuse individually (7/10) or within their organization (5/10), but often did not consider this substantial enough to be 'tooling' (5/10). Five interviewees thought further tooling would be useful, while four were skeptical it could provide sufficient microarchitectural control. A further challenge was losing access to one's code or tooling when switching organizations (2/10).

## 3.6 RQ4: Development Process and the Role of Prototyping

When asked directly, 6/10 of our interviewees claimed there was no process common to the attacks they had worked on. However, as they described their work to us we identified several process commonalities, along with two distinct development scenarios. In scenario *A*, work focuses on identifying or implementing an entirely novel side channel, entailing substantial experimentation. In scenario *B*, a side channel is known and the focus is on identifying attack targets for exploitation. Our focus in this work is facilitating manual attack discovery and PoC creation. Hence, we only analyze the development process for scenario *A*; orthogonal tooling exists for scenario *B* [131, 132, 134].

If not prompted by a disclosure, the development process starts with an initial idea or observation. Reported sources of ideas included manufacturer documentation, patents, knowledge of prior art, research discussions, blog posts, and teaching. Experimentation with microbenchmarks and implementation of a simple PoC then begins. Our interviewees stressed the importance of having as much control over the microarchitecture as possible when starting with a PoC (9/10). Even if aiming to produce a browser-based attack, one would always begin with native code. Many of our interviewees found this implementation stage challenging (6/10). However, this was attack-dependent, with three interviewees reporting that initial prototyping was straightforward and rapid ($<$ 24 hours).

Throughout the development process, communication of ideas is necessary within the team and with third parties, e.g., for coordinated disclosure. For our academic interviewees, this included collaborating on a paper and negotiating the potential conflict between disclosure embargoes and their publication requirements. Our industry interviewees needed to negotiate with management and align mitigation strategies with other industry players, while taking extensive measures to maintain the embargo and avoid allegations of collusion. In

| **Library** | **Mastik** | **cacheutils** | **libsc** |
|---|---|---|---|
| There are a good range of high-level features | 69.6% | 66.7% | 85.7% |
| There are a good range of low-level features | 56.5% | 62.5% | 71.4% |
| It was easy to understand the library at the beginning | 34.8% | 66.7% | 90.5% |
| Once I had familiarized myself with the library, it was easy to use | 69.6% | 87.5% | 90.5% |
| The function names made it clear how they should be used | 43.5% | 79.2% | 85.7% |
| It would be easy for me to adapt or extend the library if I needed to | 30.4% | 66.7% | 61.9% |
| The library was easy to build | 17.4% | 100% | 90.5% |
| I think this library makes it faster to implement attacks compared to developing from scratch | 65.2% | 87.5% | 81.0% |
| I think this library reduces the amount of debugging needed compared to developing from scratch | 47.8% | 66.7% | 71.4% |
| I spent a lot of time debugging my code because I misunderstood the library functions or they didn't do what they were supposed to | 17.4% | 8.3% | 19.1% |
| The documentation and examples were helpful | 26.1% | 41.7% | 66.7% |
| The library has sufficient documentation and examples | 26.1% | 33.3% | 61.9% |

Table 2: Summary of the user study results evaluating the usability of cacheutils, Mastik and libsc.

both settings, PoCs play an important role in communication. For example, manufacturers normally request PoCs in the disclosure process [42], and for mitigations, PoCs help assess the risk posed by a vulnerability, as one of our industry interviewees reported: *"[I wrote] an exploit that leaks 100KB/s...that really completely changed the discussion internally."*

Once initial PoCs have been created, if the goal is an end-to-end attack then the assumptions made can gradually be reduced until the attack model is realistic. However, rather than a linear progression from experimentation to a 'deliverable' such as an end-to-end attack and paper, or a full product mitigation, interviewees described an iterative process (5/10). There are clear parallels with the rapid prototyping methodologies used in software development: while an end-to-end attack may be iteratively developed from an initial prototype (5/10), *i.e.*, is an *evolutionary* prototype, the process may also involve many other microbenchmarks and experiments, *i.e.*, *throwaway* prototyping [34].

**Current Limitations of Prototyping.** Section 2.3 introduced the different use cases for prototypes throughout the product development process. Similarly, PoCs - the rapid prototypes of the security world - serve many of these roles in microarchitectural attack development. However, typical PoCs have limitations versus the rapid prototypes produced in product development. In particular, 7/10 of our interviewees highlighted that a PoC is typically only useful for communication on a surface level, e.g., demonstrating a vulnerability to collaborators or management. Microarchitectural attack code can be difficult for even experienced practitioners to comprehend (3/10) and is extremely fragile. One interviewee also stressed the importance of others independently reimplementing PoCs to ensure the root cause is properly understood.

### 3.7 Summary

We conclude from our analysis that most practitioners informally follow an iterative development process. Such a process is ideally suited to rapid prototyping [34, 69]. However, the pace of prototype development is currently slowed by substantial software implementation challenges that could be facilitated by improved tooling. While existing tools have helped facilitate attack research, none provide all common attack building blocks on their own, and they have very different interfaces and intended use cases. A further challenge is that PoCs are less effective as communication tools than traditional prototypes. A high-level API could enable creation of PoCs that are more expressive and easier to understand, helping to abstract away from implementation details to communicate and reason about the underlying concepts.

## 4 Frameworks

Informed by our findings in Section 3, we developed two frameworks to support rapid prototyping in the microarchitectural attack development process. The first, *libtea*, implements the fundamental microarchitectural attack building blocks identified for RQ2 in the most widely-used attack development language, C. The second, *SCFirefox*, brings the capabilities of *libtea* to JavaScript in Firefox to facilitate porting of native-code prototypes to the browser. Both frameworks, including API documentation and examples, are available on GitHub [28].

### 4.1 *libtea*

*libtea* (short for *Transient Execution Attack library*) provides fine-grained control of microarchitectural behavior via a high-level, platform-agnostic API. The API blurs the traditional boundaries between unprivileged C code, the OS, and the TEE (Intel SGX) by mapping privileged interfaces into user-space. It combines the functionality of *PTEditor* and *SGX-Step* with a revised and extended version of our *libsc* library (informed by the feedback from our user study) in a unified API. As previously shown in Table 1, it provides all of the attack building blocks identified in RQ1. This facilitates rapid prototyping, as cross-platform attacks requiring control over OS features such as page tables, scheduling, and interrupts can be developed without needing to modifying the kernel or consider platform-specific implementation details. Privileged functionality is provided by a single kernel driver, enabling attacks to be prototyped within a mainstream OS (Linux or Windows) to take advantage of its rich application environment.

**Implementation.** *libtea* is implemented in C as a collection of files that are configured and compiled into a single modular C header and a supporting kernel driver. The header is designed to be used in combination with other software libraries and is highly configurable. Unused modules can easily be removed in the build process, and all names and definitions are prefixed to minimize the risk of namespace conflicts. Architecture-specific code is in separate files so that adding support for a new architecture is straightforward and does not require edits across the entire codebase. The kernel driver enables functionality that requires privileged instructions or close interaction with the OS, for example modifying page tables.

Our identified attack building blocks are provided across five modules and described in detail in Appendix A. *Common* contains general primitives, e.g., high-resolution timers and functions to reduce system noise. *Cache* provides cache-attack primitives, e.g., Flush+Reload and Prime+Probe, and high-level functions such as cache covert channel encoding and decoding. *Paging* provides functionality to interact with page tables and page-translation caches. These first three modules support Linux and Windows 10 on x86-64, as well as Linux and Android on ARMv8-A. *Common* additionally supports Linux on PPC64. Note that the *Cache* and *Common* modules can be used without the kernel driver with the exception of a few privileged functions. It is required, however, for typical use cases of the *Paging*, *Interrupts* and *Enclave* modules. The primitives in these modules are intended for privileged attack scenarios (in particular, attacking Intel SGX), as they cannot be replaced with unprivileged alternatives. *Interrupts* and *Enclave* are built upon *SGX-Step* and currently only support Linux on x86-64[2]. In addition to functionality for manual interrupt and call gate configuration, *Interrupts* provides a helper function to easily execute arbitrary C functions in kernel mode. This facilitates rapid prototyping when short snippets of privileged code are required, as these can be modified and tested without needing to recompile and reload a kernel driver.

## 4.2 *SCFirefox*

*SCFirefox* facilitates browser-based attack research by blurring the boundaries between sandboxed JavaScript and privileged native code in a first-of-its-kind framework that exposes the *Common*, *Cache*, and *Paging* functionality of *libtea* to JavaScript in a modified Firefox browser. This functionality greatly facilitates rapid prototyping and eliminates the need to manually modify the JavaScript engine for basic attack building blocks, such as high-resolution timers as in prior work [66, 123]. For example, a pointer to memory with the 'present' bit cleared (to prototype a Foreshadow or RIDL PoC) can be obtained in one line of JavaScript with SCFirefox, rather than having to determine the memory layout of the browser tab's process and clear the bit in a separate native code program using e.g., *PTEditor* or *SGX-Step*. This enables rapid prototyping of gradually more realistic PoCs, quickly producing an initial PoC with *SCFirefox* and then iteratively replacing building blocks with vanilla JavaScript or WASM, as we demonstrate in Section 5.3.

Facilitating browser-based attack research is crucial for our understanding of the threat posed by microarchitectural attacks, as an attack in the browser has far greater potential impact. However, there are no existing frameworks to facilitate the creation of browser-based PoCs, and such PoCs are more challenging to implement than in native code due to the absence of many common building blocks [83], e.g., cache-line flushing and high-resolution timers [36, 101, 127], and recent mitigations implemented by browser vendors against side-channel and transient execution attacks [121]. While there are JavaScript libraries for exploit development [86], these focus on memory corruption exploits rather than microarchitectural attacks, and to our knowledge none provide out-of-the-box support for performing kernel-mode operations (e.g., modifying page-table entries) directly in JavaScript.

**Implementation.** *SCFirefox* is implemented as a custom `JSClass` within the Firefox JavaScript engine, SpiderMonkey, using the engine's C++ API, `JSAPI` [75]. The current release is compatible with Firefox 89, but the modified files are relatively stable since Firefox 81, so we anticipate that porting it to newer versions is feasible. The *SCFirefox* API can be used either directly in the browser or in the SpiderMonkey shell.

## 4.3 Rapid Prototyping with the Frameworks

As we learned from our expert interviewees, the development process varies for each attack, and each team may prefer a different approach to prototyping with our frameworks. However, the following is an overview of how one might rapidly prototype a browser-based attack. Section 5.3 provides a concrete example of this process being used to prototype the first browser-based ZombieLoad attack.

**Stage 1:** initial prototyping with *libtea* begins with experiments to test a hypothesis (e.g., 'under condition X, data can be leaked from A') or, if working with an existing attack, by reproducing an existing native code PoC.

**Stage 2:** throwaway prototypes are built with *libtea*. These may be further experiments to better understand the leakage source and the conditions necessary to induce leakage, or attempts to improve the leakage rate or relax the assumptions made about the attacker's capabilities.

[2]It is unfortunately not possible to port the SGX single-stepping functionality to Windows (without relying on undocumented APIs that may vary between OS builds), because the hardware abstraction layer does not provide drivers direct access to the local APIC timer [139].

**Stage 3:** evolutionary prototyping of the final attack begins in *SCFirefox*. *SCFirefox* API calls are replaced one by one with vanilla JavaScript or WASM until the attack can be conducted in an unmodified browser.

**Limitations.** Even when using our frameworks, certain implementation challenges remain, such as those unique to microarchitectural attacks in JavaScript [11,36,39,58,83,99,101,123]. These include ensuring that JavaScript code is indeed JIT compiled and not interpreted or optimized away [17,18], triggering misspeculation, and avoiding garbage collection [72]. We discuss some of these challenges in Section 5.3. Due to varying branch prediction implementations, it is particularly challenging to implement a misspeculation gadget that provides a long transient window across microarchitectures, so attacks may require a hand-crafted Spectre gadget.

**Ethical Considerations.** In line with the ethical guidelines of Google's SafeSide project [33], our frameworks' aim is to facilitate attack prototyping only under controlled research conditions (e.g., in a modified browser). End-to-end attacks can indeed be created from prototypes by replacing each framework building block with an unprivileged/vanilla alternative. Crucially, however, this requires manual effort and often creation of novel attack techniques, as we demonstrate in Section 5.3. This is an important difference from, for example, penetration testing frameworks such as Metasploit [91], and means that our frameworks do not enable automated creation of end-to-end exploits that could be weaponized.

## 5 Evaluation

In this section, we demonstrate the effectiveness of rapid prototyping with our frameworks in three case studies, considering in turn the usability of the *libtea* API, its cross-platform support, and the rapid prototyping process with *SCFirefox*.

### 5.1 API Usability

To evaluate the usability of the unified API provided by *libtea*, we prototype a Foreshadow PoC on x86 using the *libtea* C header and kernel driver. We consider an educational scenario where a student prototypes a Foreshadow attack against an SGX enclave and attempts to experimentally demonstrate that (counter-intuitively) the attack will succeed *even when the enclave secret is marked as uncacheable*. This occurs because the memory type specified for the Processor Reserved Memory used by SGX overrides other memory types, such as those from the Page Attribute Table (PAT) [51].

Using existing tooling, this PoC would require both *PTEditor* and *SGX-Step*, because *SGX-Step* has no support for modifying the PAT to make a page uncacheable, while *PTEditor* has no support for SGX enclaves. However, while these two frameworks can be used together, they have extremely different APIs. In particular, they modify the page tables differently with two separate drivers, so it requires care and a detailed understanding of paging to avoid crashes. *libtea* unifies the two APIs so that practitioners can more easily benefit from the functionality of both frameworks. We took care to ensure that method names are as self-explanatory as possible to improve usability, and to provide additional helper functions to abstract away from implementation details that can slow development due to the cognitive overhead and debugging required. For example, for improved readability and to abstract away from this Intel-specific implementation detail, the `edbgrdwr` function from *SGX-Step* is renamed `libtea_{read,write}_secure_addr`. Similarly, we provide a single `libtea_set_page_cacheability` function as opposed to the combination of `ptedit_find_first_mt` and `ptedit_apply_mt` required in *PTEditor*.

The abstraction offered by *libtea* is further illustrated in Listing 1, which compares the code required using each of the three frameworks to obtain a secondary mapping for Foreshadow. In this example, we make the same assumption as in the *SGX-Step* Foreshadow PoC [113] that we have the address of an enclave secret at `enclave_ptr`. Any architectural access from outside the enclave (when running in release mode) will trigger abort page semantics (returning $-1$), but if the 'present' bit is cleared then we can transiently obtain the data with Foreshadow. Obtaining a second virtual mapping $v2$ to the underlying physical address allows us to clear the 'present' bit only on $v2$, so that the architectural enclave access still succeeds. Calling `mprotect` with `PROT_NONE` will clear the 'present' bit, but will change the PFN to point to an uncacheable address. This is due to the PTE inversion mitigation for Foreshadow, which cannot be disabled [20]. Clearing the bit manually instead (e.g., with *libtea*) will confuse the kernel and can lead to the process being killed or even a system crash. Instead, we must restore the unmitigated PFN after the `mprotect` call informs the kernel this page is now unmapped. Implementing this with *PTEditor* or *SGX-Step* requires a good understanding of paging, and any mistake in the implementation may cause a system crash. In contrast, *libtea* handles this complexity for the user so that just two lines of code are necessary.

Our Foreshadow PoC using TSX for exception suppression requires 48 lines of code with 12 *libtea* calls. On an Intel i7-6700K with unmitigated microcode we achieve a 99.8 % mean success rate over 1000 trials across all 64 cache lines of a 4KB enclave page. Alternatively, using exception handling we require 46 lines of code and 10 *libtea* calls, achieving a success rate of 92.6 %. With an additional 22 lines and 10 *libtea* calls we can repeat the attack with an 'uncacheable' $v1$ and $v2$, where we observe a comparable success rate with TSX (99.4 %) because, as discussed, SGX ignores the memory type specified in the PAT. Note that we use an enclave in debug mode for all three PoCs, as a commercial license agreement is required to launch in release mode.

```
//Foreshadow mapping with SGX-Step
void* alias_ptr = remap_page_table_level(enclave_ptr, PAGE);
void* pte_alias = remap_page_table_level(alias_ptr, PTE);
uint64_t pte_alias_unmapped = MARK_NOT_PRESENT(*pte_alias);
mprotect((void*) (((uint64_t) alias_ptr) & ~PFN_MASK), 4096,
      PROT_NONE);
*pte_alias = pte_alias_unmapped;

//Foreshadow mapping with PTEditor
size_t pfn = ptedit_pte_get_pfn(enclave_ptr, 0);
void* alias_ptr = (void*)ptedit_pmap(pfn * 4096, 4096);
mprotect((void*) (((uint64_t) page) & ~0xfffULL), 4096,
  PROT_NONE);
ptedit_pte_set_pfn(alias_ptr, 0, pfn);

//Foreshadow mapping with libtea
void* alias_ptr = libtea_remap_address(instance, (uint64_t)
      enclave_ptr, LIBTEA_PAGE, 4096, PROT_READ, true);
libtea_mark_page_not_present(instance, alias_ptr);
```

Listing 1: Comparison of using the three frameworks to obtain a secondary mapping for Foreshadow.

## 5.2 Cross-Platform Support

To evaluate the effectiveness of the cross-platform support provided by *libtea*, we prototype the first Load Value Injection (LVI) attack on an ARMv8-A SoC using the *libtea* C header and kernel driver. Our target device is a Samsung Galaxy S7 Edge smartphone running Android 7.0 with kernel 3.18.14. It uses a Samsung Exynos 8890 SoC with four Cortex-A53 cores and four Exynos Mongoose 1 (M1) cores. We prototype an LVI-US-L1D PoC leaking 2.49 kB/s ($n$ = 10000, $\sigma_{\bar{x}}$ = 0.63) on our target device.

**Attack Scenario.** Van Bulck et al. [115] did not include LVI-US-L1D in their classification tree, concluding that it cannot be used because *"a benign victim process would never dereference kernel memory"*. However, while a benign process is indeed unlikely to *architecturally* dereference kernel memory, we consider the scenario where a benign process *transiently* dereferences a kernel memory address due to misspeculation. In line with *libtea*'s intended purpose, our LVI-US-L1D attack is a prototype and not a full end-to-end attack. However, an end-to-end LVI-US-L1D attack may be feasible, even on systems with active Meltdown mitigations. The key requirements are direct or indirect attacker control over a speculatively dereferenced address and the presence of an LVI gadget within this transient execution stream. For example, program data (e.g., a negative 64-bit integer variable) might be confused as a kernel address in a misspeculated branch. If the attacker controls this data, they control which address is speculatively accessed and can therefore inject data.

On systems without in-silicon fixes for Meltdown, the KPTI patch [29] is required to mitigate Meltdown. While this unmaps most kernel addresses from user-space, some kernel addresses must necessarily remain mapped. Therefore, an attacker could still inject values from the small set of kernel addresses which remain mapped into each user process. These provide a variety of data values that could be injected into the victim, particularly if an attacker only needs to inject a single byte, as there is a good probability of finding the required byte value 0-255 within the at least 4096 B of mapped kernel memory.

**Attack Development Process.** The Exynos M1 cores are known to be susceptible to Meltdown [67]. We verified this susceptibility to Meltdown using *libtea*, suppressing exceptions via misspeculation and encoding values into a Flush+Reload-based covert channel. Building from this PoC, we conducted throwaway prototyping investigating the M1's susceptibility to Foreshadow and the MDS vulnerabilities. This was ideally suited for rapid prototyping as each prototype required changing only a few lines of code, e.g., to clear a different bit in a page-table entry (PTE) using the privileged page-table functionality provided by the kernel driver. We did not observe leakage with any of these prototypes, and conclude that the M1 is likely not vulnerable to Foreshadow or any MDS variant. However, we were able to invert Meltdown to mount an LVI attack (*i.e.*, LVI-US-L1D).

**LVI-US-L1D PoC.** In our toy scenario, the victim dereferences an array of attacker-provided values. The values within the loop bound are valid user-space addresses, while the values beyond the loop bound are malicious kernel-space addresses that are transiently dereferenced due to misspeculation. We choose these addresses to be direct-physical map addresses for the attacker's own process, and can therefore inject any byte value(s) we wish. We rely on root privileges to determine the kernel direct-physical map address of our toy injected data, but this is not necessary for an end-to-end exploit [67]. For evaluation purposes, we consider a simple cache encoding LVI gadget to recover our own injected bytes.

```
void victim() {
  size_t dummy = 0;
  uint64_t c = pow(2, 63);
  for(int i = 0; i < 100; i++) {
    /* Increase size of transient window */
    dummy = 1; libtea_flush(&dummy); libtea_access(&dummy);
    /* Mistrain branch prediction */
    for (volatile int u = 0; u < 100; u++) {
      asm volatile ("nop");
    }
    libtea_flush(&c);
    /* Loop targeted with Spectre V1 */
    if(c / 0.5 > 1.1) {
      /* Transiently dereference value at i>=100 */
      char val = *addrs[i];
      /* LVI disclosure gadget */
      libtea_cache_encode(instance, (val - 'A'));
    }
    c /= 2;
  }
}
```

Listing 2: Toy victim function for LVI.

Listing 2 shows our toy victim function, which contains a loop vulnerable to Spectre V1. Within the loop, the attacker-controlled array of addresses is dereferenced, and the dereferenced values are encoded into the cache by a disclosure gadget (`libtea_cache_encode`). We can suppress exceptions by providing valid user-space addresses for the array indices within the loop bound and only including the kernel-space address used for injection at indices beyond this bound. Thus, these out-of-bounds addresses are only ever accessed transiently due to misspeculation, and any value checks that the victim might conduct before the loop do not apply. To extend the transient window, we flush a `dummy` variable from the cache and then access it so that the CPU will speculate beyond this point while it waits for the load to complete. To further extend the window, we include two pages flushed from the cache (`throttle`) in our loop condition. Misspeculation occurs because the loop condition is met while $i < 64$, training the branch predictor to continue with the next iteration. Hence, the processor misspeculates for multiple iterations although the condition is architecturally false.

Our prototype consists of an injector process and an attack process, both running on the same Exynos M1 core. The injector process continuously accesses the byte to inject to keep it in L1. The attack process makes an API call to the toy victim function, decodes the injected data using Flush+Reload, and outputs the decoded values as a histogram. This can be achieved in a single function call using *libtea*'s cache covert channel histogram functions, as shown in Listing 3.

```
libtea_instance* instance = libtea_init_nokernel();
/* Set up attacker-controlled array of addresses */
for(int i = 0; i < 64; i++)
  addrs[i] = valid_addr;      //architecturally accessed
for(int i = 64; i < 100; i++)
  addrs[i] = kernel_inject_addr; //transiently accessed
libtea_set_timer(instance, LIBTEA_TIMER_NATIVE);
libtea_print_cache_decode_histogram(instance, 100000,
  50, true, true, victim, 'A', 0, 26);
libtea_cleanup(instance);
```

Listing 3: Exploiting the toy function with *libtea*.

**Evaluation.** To evaluate the attack, we averaged the results over 10 000 iterations of 10 000 samples. We also evaluated the prototype on three Intel CPUs (i7-8700K, i7-6700K, and i7-4790). Aside from retuning the covert channel threshold, no changes were necessary to port the PoC to x86 due to the cross-platform support of *libtea*. However, it is important to note that different CPU microarchitectures can exhibit very different branch prediction behavior. Fundamental microarchitectural differences such as this cannot be abstracted away by *libtea*, and so an attack may require or benefit from microarchitecture-specific tuning. For example, by modifying our Spectre V1 gadget to use throttle variables in the loop condition we can achieve an improved leakage rate of 3.91 kB/s on the Exynos M1, whereas on the Intel CPUs this change actually inhibits the leakage.

Table 3 compares the leakage rates and $F_1$ scores achieved on the M1 and the Intel microarchitectures. With around 7 kB/s, the leakage rates we achieve on Coffee Lake and Skylake are comparable to the ideal case leakage reported by Van Bulck et al. [115] (9.04 kB/s) and substantially exceed their leakage in other attack scenarios (e.g., 70.54 B/s). Our high $F_1$ scores on both ARMv8-A and x86 demonstrate not only that *libtea* can be used to prototype low-noise microarchitectural attacks, but also more generally that it is indeed possible to prototype cross-platform attacks using a high-level library.

| **CPU** | **Leakage (kB/s)** | **$F_1$ Score** |
|---|---|---|
| Exynos M1 | 2.49 | 0.98 |
| i7-8700K (Coffee Lake) | 7.36 | 0.99 |
| i7-4790 (Haswell) | 7.20 | 0.98 |
| i7-6700K (Skylake) | 7.46 | 0.99 |

Table 3: LVI leakage across the evalauted microarchitectures.

## 5.3 Rapid Prototyping with *SCFirefox*

In this section, we evaluate *SCFirefox* by rapidly prototyping a browser-based ZombieLoad attack. Specifically, we prototype ZombieLoad variant 3, as this is the only variant that can be conducted by an unprivileged attacker without access to TSX [100]. To fully illustrate rapid prototyping in action, we provide a detailed description of our development process. *Note: throughout this section, we use ⓝ to denote specific attack building blocks.*

*Experimental Setup.* The experiments described in this section were conducted on an Intel i7-8700K running Windows 10 Pro v2004 (build 19041.388) with microcode 0xB4 and full transient-execution attack mitigations. For a more stable core frequency, we disabled Intel SpeedStep and Intel Turbo Boost. We also conducted our accessed bit clear experiments on two other systems (Windows 10 Enterprise v1803 build 17134.1, i5-4300; Pro v1909 build 18363.1082, i5-6400).

**Technical Background.** For ZombieLoad variant 3, the attacker requires two mappings (virtual addresses) *v1* and *v2* to the same physical address ①. If the PTE of *v1* has the 'accessed' bit cleared ②, a microcode-assisted page-table walk will occur when accessing the physical page through this address. Simultaneously flushing *v2* leads to a cache-line conflict ③ and can induce data leakage from the line fill buffer and load port. This data can be transiently encoded for subsequent decoding, e.g., in a cache covert channel ④.

*Working set trim scan.* Windows clears the 'accessed' bit during the working set trim scan to track the 'age' of each PTE [139]. The working set manager is called once per second and additionally when certain memory conditions occur [139]. However, it does not always conduct a trim scan, and it is not documented how often this occurs.

*Page deduplication.* Page deduplication, known as page combining on Windows and Kernel Same-Page Merging on Linux, scans memory for identical pages, releases all but one of the pages, and marks the remaining pages as shared. If a process writes to this shared page, a copy-on-write fault creates a new copy of the page for the process [139].

**Stage 1: Native Code PoC.** We begin by reproducing ZombieLoad variant 3 in native code on Windows using *libtea*. All four building blocks are provided by *libtea* functions. For ②, we require the *libtea* kernel driver to modify PTEs, while for ③ we induce a cache-line conflict by flushing *v2* immediately before accessing *v1*. To achieve a practical leakage rate, we construct a misspeculation gadget to speculatively access *v1* and encode its value into our *libtea* cache covert channel ④ without setting the 'accessed' bit, so that we only need to clear it once. We favor misspeculation over exception handling or TSX because we cannot use the latter two techniques in a JavaScript PoC. Schwarz et al. [100] achieved a leakage rate of 0.08 kB/s on an i7-8650U using exception handling. We achieve leakage rates of 0.15 kB/s ($n$ = 10000, $\sigma_{\bar{x}}$ = 0.002), exceeding the public PoC, with a low false positive rate (0.002%). This demonstrates that *libtea* enables prototyping of highly efficient PoCs. The number of lines of code required is comparable to the public PoC, which uses *cacheutils* (75 lines of code versus 61 for our PoC). We found that cursor movement and interaction with the GUI are necessary for continued leakage with our PoC; when idle, leakage begins to fail after around 1000 iterations. This does not appear to be caused by interrupts or polling for physical mouse input, as we also tested with software-injected cursor movement.

**Stage 2: Throwaway Prototyping.** We conduct throwaway prototyping with *libtea* to determine if we can implement our building blocks using only unprivileged code. If we cannot, then an attack in an unmodified browser is infeasible.

*Clearing the 'accessed' bit without libtea.* While an unprivileged attacker cannot modify PTEs (without an additional exploit, e.g., Rowhammer [39]), Schwarz et al. [100] experimentally determined that Windows 10 periodically clears the 'accessed' bit on user pages.[3] If we map a memory page, access it, and then repeatedly loop checking the 'accessed' bit of its PTE (using the *libtea* paging functions) without accessing the page itself, we observe that the bit is first cleared on average after 0.50s ($n$ = 100, $\sigma_{\bar{x}}$ = 0.09). We confirmed this behavior on our other two systems, observing average timings of 152.27s ($n$ = 50, $\sigma_{\bar{x}}$ = 89.45) and 1.45s ($n$ = 50, $\sigma_{\bar{x}}$ = 0.10) respectively. While the timing appears to vary by Windows build and system specification, in all three cases the time required is feasible for a browser-based attack.

*Obtaining shared memory.* ZombieLoad variant 3 requires shared memory [100], which is not readily available in JavaScript. In the Firefox codebase, we did not find any code we could trigger from JavaScript that uses `CreateFileMapping`. On Windows, an address can only be mapped multiple times (creating multiple 'views') if it is a file mapping opened with this function. However, we observe that despite formerly being disabled as a mitigation [36], page combining is reenabled on Windows 10 by default, enabling us to revive prior browser-based page deduplication attacks [11, 36, 37] to obtain shared memory. We rely on deduplication within our own process to obtain two virtual addresses mapping to the same physical address. For prototyping, we implement a new function for *libtea* and *SCFirefox* to manually trigger a page combining scan and rapidly achieve deduplication. However, a limitation versus our native code shared memory is that we cannot write to either address.

*Inducing a cache-line conflict without* `clflush`. A common strategy in prior work is to replace flushing with eviction when porting attacks to the browser [36, 39, 58, 83, 95, 101]. However, it is unlikely that any eviction set would evict the cache line containing *v2* at precisely the same moment as the microcode assist for *v1*, which is required for ZombieLoad [100]. Fortunately, by prototyping with *libtea*, we discover that simply *accessing* a different offset within the same cache line is sufficient to trigger a conflict and therefore leakage. However, the leakage rate with an access-based conflict is significantly lower than with flushing. While we achieve a leakage rate of 0.15 kB/s with flush-based conflict in native code, we only achieve 2.55 B/s with access-based conflict in native code.

**Stage 3: Evolutionary Prototyping.** Having determined how to implement our building blocks, we begin prototyping our attack in *SCFirefox*. Porting the PoC is mostly straightforward, as non-transient *libtea* API calls can simply be replaced with *SCFirefox* calls. When porting ③ we find that in JavaScript we also have to randomly vary the offset we access *v2* at so that the JIT does not optimize the access away.

*Prolonging the Transient Window.* The final challenge to produce an *SCFirefox* PoC is to create a sufficiently long transient window. Browser-based misspeculation was first demonstrated by Kocher et al. [58] with a transient out-of-bounds array access. Due to Spidermonkey's Spectre mitigations, such out-of-bounds accesses are no longer possible. However, Spectre is not entirely mitigated and we can conduct our attack with an in-bounds transient array access. We found that most strategies developed in prior work [58, 70, 95, 123] did not provide a sufficiently long transient window for ZombieLoad variant 3 on the i7-8700K, but succeeded using a gadget adapted from the Tencent PoC [112].

Our full ZombieLoad variant 3 PoC in *SCFirefox* consists of 392 lines of code, relying on 15 *libtea* API calls. The majority of these code lines are dummy lines required for branch mistraining or to prevent function inlining. We achieve a leakage rate of 55.85 B/s ($n$ = 1000, $\sigma_{\bar{x}}$ = 12.43).

Finally, we replace *SCFirefox* functions with JavaScript and WASM functions to produce a PoC for unmodified Firefox.

*Page Combining.* Instead of manually triggering deduplication for ①, we adopt the technique used by Gras et al. [36] to detect when a page-combining scan occurs. We fill 8 4 kB `ArrayBuffers` with identical random content and measure

[3]Note that the Linux kernel swap daemon also clears the 'accessed' bit for page aging on x86 (see `arch/x86/mm/pgtable.c`). However, in our experiments we did not succeed in triggering an 'accessed' bit clear.

the time it takes to modify a value of the array: when it takes significantly longer, we assume we have encountered a copy-on-write fault. We experimentally confirmed that our pages are deduplicated, and found that it takes on average 12.49 minutes ($n = 25$, $\sigma_{\bar{x}} = 4.43$, range: $4.58 - 22.39$ minutes). Similarly, we cannot clear the 'accessed' bit, but we can wait for Windows to clear it. We find that the 'accessed' bit is set on *v1* after page combining, but within a few attack iterations (1-18) it is again cleared.

*Timing.* For our cache covert channel ④ we replace *SCFirefox*'s Flush+Reload with Evict+Reload, using a pseudorandom buffer traverse for eviction (to evade adaptive cache replacement policies [105]). We experiment with various JavaScript timers [36, 101, 127]. While `SharedArrayBuffer` has been re-enabled in Firefox [121], it provides insufficient precision when used in a counting thread with a JavaScript Worker. However, adapting the WASM timer of Vila et al. [127] provides a sufficiently high resolution.

Our final PoC in vanilla JavaScript and WASM is 490 lines of code and runs in unmodified Firefox 81.0.2 on Windows. As using Evict+Reload and a WASM Worker timer introduces some noise versus our *SCFirefox* PoC, we consider the three most frequently leaked values per attack iteration of 200 samples as candidates for the leaked byte. In line with previous work [36, 100, 101, 123], we assume that we can repeatedly leak the target values. Hence, any noise is averaged out over sufficient attack repetitions. We achieve a leakage rate of 1.48 B/s ($n = 1000$, $\sigma_{\bar{x}} = 0.99$), top-3 accuracy of 92.7 % and a true-positive rate (leaked byte is indeed the most frequently leaked value) of 88.8 % when our preconditions ('accessed' bit clear, deduplication, co-location) are met. This is comparable to the RIDL attack in a modified SpiderMonkey shell [123] which leaked 1 B/s. Furthermore, if we replace Evict+Reload with Flush+Reload using *SCFirefox*, we can achieve a much higher leakage rate of 10.68 B/s ($n = 1000$, $\sigma_{\bar{x}} = 6.48$). This demonstrates that there is scope to optimize our Evict+Reload-based PoC, e.g., by using a minimal eviction set [127].

**Attack Scenario and Discussion.** In our attack scenario, the victim must browse to an attacker-controlled webpage that remains open for 5-23 minutes, depending on when the next page combining scan occurs. However, the tab does not need to be in the foreground during this time: we confirmed that the attack succeeds even while the victim is browsing in other tabs. A final requirement is to synchronize with the victim so that the attack runs while the targeted data is passing through the line fill buffers; this can be achieved via one of the synchronization techniques presented by Van Schaik et al. [123] and Schwarz et al. [100], e.g., substring detection if the secret is prefixed by a known sequence.

A significant assumption we make for evaluation is that we have achieved co-location on the sibling core to the victim. Without pinning, the Windows scheduler reschedules the browser tab thread to different cores so frequently that attempting to detect co-location to synchronize the attack is infeasible. Despite this, in the ideal case with victim applications running on one hyperthread of every physical core, we can still achieve a leakage rate of 1.18 B/s ($n = 1000$, $\sigma_{\bar{x}} = 0.72$). However, we observe close to zero leakage in the more realistic scenario of a single victim application running pinned to one core. Future work could investigate methods to influence the scheduler's behavior from JavaScript to achieve co-location, e.g., this might be an alternative application for thread spraying [32].

## 6 Discussion

One limitation of our analysis of the development process is the small scale of our expert interview study and the restricted demographics of our user study. Many industry teams consider their development process confidential, and as such there is limited publicly-available information and reluctance to approve interviews. Furthermore, in contrast to areas such as software reverse-engineering [128], the field of microarchitectural attacks is still small. Our aim in conducting our user study among students of an introductory microarchitectural security course was to complement our data from experienced practitioners with data from beginners to the field. However, the resulting homogeneity in participants' academic background (21/28 were studying for a Master's degree in Computer Science or a related field) may have biased our results, along with the homogeneity in gender and age (all participants were male, with 24/28 aged $23 - 26$). Additionally, we could not evaluate the usability of *libtea* because we conducted the user study at the beginning of the design process.

We consider this work a first step towards establishing concrete methodology and tooling for microarchitectural attack development, and hope future work will further explore attack tooling usability, techniques for achieving microarchitectural control, and challenges beyond software implementation. One such challenge discussed by our interviewees was the disclosure process (7/10) and potential standardization and anonymous disclosure mechanisms. While this challenge is not unique to microarchitectural security, the cross-cutting nature of microarchitectural attacks has led to notably challenging disclosure experiences that highlighted the need for greater standardization [12, 21, 30]. Such work would also be of broader benefit to other information security practitioners.

There is also scope to further extend our frameworks, for example integrating with existing software tools such as `perf` or NanoBench [1] to provide access to hardware performance counters for microbenchmarking. Additionally, *libtea* could be extended to provide fine-grained control of other TEEs such as Arm TrustZone, while *SCFirefox* could be ported to other JavaScript engines, e.g., the V8 engine used by Google Chrome and Microsoft Edge.

## 7 Conclusion

The complexity of microarchitectural attack implementation poses a high barrier to entry for the field and slows research progress. In this paper, we conducted the first investigation of the attack development process to determine how we could best help practitioners tackle this complexity. We introduced rapid prototyping as an attack development methodology and presented *libtea* and *SCFirefox*, open-source frameworks which facilitate rapid prototyping of native code and browser-based attacks by bridging traditional privilege boundaries and abstracting away from the complex implementation details of attack primitives. We demonstrated the improved usability offered by the *libtea* API with a Foreshadow PoC, showcased its cross-platform support with the first demonstration of LVI on ARMv8-A, and illustrated the benefits of *SCFirefox* and our rapid prototyping methodology by prototyping the first browser-based ZombieLoad attack.

## Acknowledgments

We would like to thank our anonymous reviewers and in particular our shepherd, Chris Fletcher, for their feedback that helped improve this paper. We also thank Martin Deixelberger for his work on *SCFirefox*; Moritz Lipp, Claudio Canella, and Jo van Bulck for helpful discussions; and our expert interviewees and user study participants for their valuable insights. This work was supported by generous gifts from Red Hat, Arm, Amazon, and Cloudflare. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties or of the authors' affiliations.

## References

[1] Abel, A., and Reineke, J. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In *ISPASS* (2020).

[2] Aciiçmez, O., Brumley, B. B., and Grabher, P. New Results on Instruction Cache Attacks. In *CHES* (2010).

[3] Aciiçmez, O., and Koç, c. K. Trace-Driven Cache Attacks on AES. *IACR Cryptology ePrint Archive* (2006).

[4] Aciiçmez, O. Yet Another MicroArchitectural Attack: Exploiting I-cache. In *CSAW* (2007).

[5] Aciiçmez, O., Seifert, J.-P., and Koç, c. K. Predicting secret keys via branch prediction. In *CT-RSA* (2007).

[6] Aldaya, A. C., Brumley, B. B., ul Hassan, S., García, C. P., and Tuveri, N. Port Contention for Fun and Profit. In *S&P* (2019).

[7] Allan, T., Brumley, B. B., Falkner, K., Van de Pol, J., and Yarom, Y. Amplifying Side Channels Through Performance Degradation. In *ACSAC* (2016).

[8] Apple Inc. About speculative execution vulnerabilities in ARM-based and Intel CPUs, https://support.apple.com/en-us/HT208394 2018.

[9] Bernstein, D. J. Cache-Timing Attacks on AES, http://cr.yp.to/antiforgery/cachetiming-20050414.pdf 2005.

[10] Biswas, A. K., Ghosal, D., and Nagaraja, S. A survey of timing channels and countermeasures. *ACM Computing Surveys (CSUR)* (2017).

[11] Bosman, E., Razavi, K., Bos, H., and Giuffrida, C. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P* (2016).

[12] Brandom, R. Keeping Spectre Secret, https://www.theverge.com/2018/1/11/16878670/meltdown-spectre-disclosure-embargo-google-microsoft-linux 2018.

[13] Bulck, J. V. cacheutils.h (linux), https://github.com/jovanbulck/sgx-step/blob/master/app/lvi/cacheutils.h 2020.

[14] Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., and Yarom, Y. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS* (2019).

[15] Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium* (2019).

[16] Chua, C. K., Leong, K. F., and Lim, C. S. *Rapid Prototyping: Principles and Applications (with Companion CD-ROM) Third Edition*. World Scientific Publishing Company, 2010.

[17] de Mooij, J. CacheIR: A new approach to Inline Caching in Firefox, https://jandemooij.nl/blog/2017/01/25/cacheir/ 2017.

[18] de Mooij, J. The Baseline Interpreter: a faster JS interpreter in Firefox 70, https://hacks.mozilla.org/2019/08/the-baseline-interpreter-a-faster-js-interpreter-in-firefox-70/ 2019.

[19] Disselkoen, C., Kohlbrenner, D., Porter, L., and Tullsen, D. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security Symposium* (2017).

[20] Documentation, L. K. L1TF - L1 Terminal Fault, https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/l1tf.html 2021.

[21] Edge, J. A look at the handling of Meltdown and Spectre, https://lwn.net/Articles/743363/ 2018.

[22] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO* (2016).

[23] Falk, B. Sushi Roll: A CPU research kernel with minimal noise for cycle-by-cycle microarchitectural introspection, https://gamozolabs.github.io/metrology/2019/08/19/sushi_roll.html 2019.

[24] Fogh, A. Behind the scenes of a bug collision, https://cyber.wtf/2018/01/05/behind-the-scene-of-a-bug-collision/ 2018.

[25] Fogh, A., and Ertl, C. Wrangling the Ghost: An Inside Story of Mitigating Speculative Execution Side Channel Vulnerabilities. In *BlackHat USA* (2018).

[26] Frigo, P., Giuffrida, C., Bos, H., and Razavi, K. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P* (2018).

[27] Ge, Q., Yarom, Y., Cock, D., and Heiser, G. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* (2016).

[28] GitHub. Libtea and SCFirefox GitHub repository, https://github.com/libtea/frameworks 2021.

[29] Gleixner, T. x86/kpti: Kernel Page Table Isolation (was KAISER), https://lkml.org/lkml/2017/12/4/709 2017.

[30] Gleixner, T. Kernel hacking behind closed doors. In *Kernel Recipes* (2019).

[31] Göktaş, E., Razavi, K., Portokalidis, G., Bos, H., and Giuffrida, C. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS* (2020).

[32] Göktaş, E., Gawlik, R., Kollenda, B., Athanasopoulos, E., Portokalidis, G., Giuffrida, C., and Bos, H. Undermining Information Hiding (and What to Do about It). In *USENIX Security Symposium* (2016).

[33] Google. SafeSide: Understand and mitigate software-observable side-channels, https://github.com/google/safeside 2019.

[34] Gordon, V. S., and Bieman, J. M. Rapid prototyping: lessons learned. *IEEE Software 12*, 1 (1995).

[35] Gras, B., Giuffrida, C., Kurth, M., Bos, H., and Razavi, K. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS* (2020).

[36] Gras, B., Razavi, K., Bosman, E., Bos, H., and Giuffrida, C. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS* (2017).

[37] Gruss, D., Bidner, D., and Mangard, S. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In *ESORICS* (2015).

[38] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS* (2016).

[39] Gruss, D., Maurice, C., and Mangard, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA* (2016).

[40] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA* (2016).

[41] Gruss, D., Spreitzer, R., and Mangard, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015).

[42] Householder, A. D., Wassermann, G., Manion, A., and King, C. *The CERT Guide to Coordinated Vulnerability Disclosure*. Carnegie Mellon University, 2017.

[43] Hu, W.-M. Reducing Timing Channels with Fuzzy Time. *Journal of Computer Security* (1992).

[44] Hund, R., Willems, C., and Holz, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P* (2013).

[45] IAIK. libflush, https://github.com/IAIK/armageddon/tree/master/libflush 2017.

[46] Intel. Deep Dive: Machine Check Error Avoidance on Page Size Change, https://software.intel.com/security-software-guidance/insights/deep-dive-machine-check-error-avoidance-page-size-change 2018.

[47] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling, 2019.

[48] Intel. Mitigations for Jump Conditional Code Erratum, Revision 1.0, 2019.

[49] Intel. Deep Dive: Snoop-assisted L1 Data Sampling, https://software.intel.com/security-software-guidance/insights/deep-dive-snoop-assisted-l1-data-sampling 2020.

[50] Intel. Vector Register Sampling / CVE-2020-0548 / INTEL-SA-OO329, https://software.intel.com/security-software-guidance/software-guidance/vector-register-sampling 2020.

[51] Intel Corporation. Software Guard Extensions Programming Reference, Rev. 2.

[52] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *S&P* (2015).

[53] Islam, S., Moghimi, A., Bruhns, I., Krebbel, M., Gulmezoglu, B., Eisenbarth, T., and Sunar, B. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium* (2019).

[54] Jang, Y., Lee, J., Lee, S., and Kim, T. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX* (2017).

[55] Kenjar, Z., Frassetto, T., Gens, D., Franz, M., and Sadeghi, A. V0LTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security Symposium* (2020).

[56] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA* (2014).

[57] Kiriansky, V., and Waldspurger, C. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).

[58] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre Attacks: Exploiting Speculative Execution. In *S&P* (2019).

[59] Kocher, P. C. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996).

[60] Koruyeh, E. M., Khasawneh, K., Song, C., and Abu-Ghazaleh, N. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT* (2018).

[61] Kwong, A., Genkin, D., Gruss, D., and Yarom, Y. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P* (2020).

[62] Lee, S., Shih, M., Gera, P., Kim, T., Kim, H., and Peinado, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium* (2017).

[63] Lipp, M., Aga, M. T., Schwarz, M., Gruss, D., Maurice, C., Raab, L., and Lamster, L. Nethammer: Inducing Rowhammer Faults through Network Requests. *arXiv:1711.08002* (2017).

[64] Lipp, M., Gruss, D., Schwarz, M., Bidner, D., Maurice, C.-m.-t.-n., and Mangard, S. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS* (2017).

[65] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium* (2016).

[66] Lipp, M., Hadžić, V., Schwarz, M., Perais, A., Maurice, C., and Gruss, D. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *AsiaCCS* (2020).

[67] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium* (2018).

[68] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-Level Cache Side-Channel Attacks are Practical. In *S&P* (2015).

[69] Luqi, and Steigerwald, R. Rapid Software Prototyping. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences* (1992).

[70] Maisuradze, G., and Rossow, C. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS* (2018).

[71] Mambretti, A., Neugschwandtner, M., Sorniotti, A., Kirda, E., Robertson, W., and Kurmus, A. Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations. In *ACM ACSAC* (2019).

[72] Mandelin, D. Know Your Engines: How to Make Your JavaScript Fast. In *O'Reilly Velocity* (2011).

[73] Masters, J., and CRob. Exploiting modern microarchitectures: Meltdown, Spectre, and other hardware security vulnerabilities in modern processors. In *Red Hat Summit* (2018).

[74] Mcilroy, R., Sevcik, J., Tebbi, T., Titzer, B. L., and Verwaest, T. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178* (2019).

[75] MDN. JSAPI User Guide, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/JSAPI_User_Guide 2020.

[76] Miedl, P., Klopott, B., and Thiele, L. Increased reproducibility and comparability of data leak evaluations using ExOT. In *Proceedings of the 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2020).

[77] Moghimi, A., Irazoqui, G., and Eisenbarth, T. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES* (2017).

[78] Moghimi, D., Bulck, J. V., Heninger, N., Piessens, F., and Sunar, B. CopyCat: Controlled Instruction-Level Attacks on Enclaves for Maximal Key Extraction. In *USENIX Security Symposium* (2020).

[79] Moghimi, D., Lipp, M., Sunar, B., and Schwarz, M. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *USENIX Security Symposium* (2020).

[80] Monaco, J. SoK: Keylogging Side Channels. In *S&P* (2018).

[81] Murdock, K., Oswald, D., Garcia, F. D., Van Bulck, J., Gruss, D., and Piessens, F. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P* (2020).

[82] Naghibijouybari, H., Neupane, A., Qian, Z., and Abu-Ghazaleh, N. Rendered Insecure: GPU Side Channel Attacks are Practical. In *CCS* (2018).

[83] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS* (2015).

[84] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[85] Page, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *Cryptology ePrint Archive, Report 2002/169* (2002).

[86] Pak, B., Wesie, A., Ahn, K. C., and Fu, S. pwn.js v1.1.0: A javascript library for browser exploitation, https://github.com/theori-io/pwnjs 2019.

[87] Percival, C. Cache Missing for Fun and Profit. In *BSDCan* (2005).

[88] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium* (2016).

[89] Qiu, P., Wang, D., Lyu, Y., and Qu, G. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *CCS* (2019).

[90] Ragab, H., Milburn, A., Razavi, K., Bos, H., and Giuffrida, C. CROSSTALK: Speculative Data Leaks Across Cores Are Real. In *S&P* (2021).

[91] Rapid7. Metasploit Framework, https://github.com/rapid7/metasploit-framework 2020.

[92] Riley, M. A Little Less Speculation, a Little More Action: A Deep Dive into Fuchsia's Mitigations for Specific CPU Side-Channel Attacks. In *Black Hat Briefings* (2020).

[93] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS* (2009).

[94] Ryan, K. Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone. In *CCS* (2019).

[95] Schwarz, M., Canella, C., Giner, L., and Gruss, D. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725* (2019).

[96] Schwarz, M., and Gruss, D. How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. *IEEE Security & Privacy* (2020).

[97] Schwarz, M., Gruss, D., Weiser, S., Maurice, C., and Mangard, S. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA* (2017).

[98] Schwarz, M., Lipp, M., and Canella, C. misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8, https://github.com/misc0110/PTEditor 2018.

[99] Schwarz, M., Lipp, M., and Gruss, D. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *NDSS* (2018).

[100] Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., and Gruss, D. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS* (2019).

[101] Schwarz, M., Maurice, C., Gruss, D., and Mangard, S. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC* (2017).

[102] Schwarz, M., Schwarzl, M., Lipp, M., Masters, J., and Gruss, D. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS* (2019).

[103] Seaborn, M., and Dullien, T. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat Briefings* (2015).

[104] Shin, Y., Kim, H. C., Kwon, D., Jeong, J. H., and Hur, J. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In *CCS* (2018).

[105] Song, W., and Liu, P. Dynamically Finding Minimal Eviction Sets Can be Quicker Than You Think for Side-Channel Attacks Against the LLC. In *RAID* (2019).

[106] Stecklina, J., and Prescher, T. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480* (2018).

[107] Sullivan, D., Arias, O., Meade, T., and Jin, Y. Microarchitectural Minefields: 4K-aliasing Covert Channel and Multi-tenant Detection in IaaS Clouds. In *NDSS* (2018).

[108] Szefer, J. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security 3*, 3 (2019), 219–234.

[109] Tang, A., Sethumadhavan, S., and Stolfo, S. CLKSCREW: Exposing the perils of security-oblivious energy management. In *USENIX Security Symposium* (2017).

[110] Tatar, A. Hammertime: a software suite for testing, profiling and simulating the Rowhammer DRAM defect, https://github.com/vusec/hammertime 2018.

[111] Tatar, A., Krishnan, R., Athanasopoulos, E., Giuffrida, C., Bos, H., and Razavi, K. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC* (2018).

[112] Tencent-Xuanwu-Lab. check.js - Spectre Check, https://xlab.tencent.com/special/spectre/js/check.js 2018.

[113] Van Bulck, J. SGX-Step Foreshadow PoC, https://github.com/jovanbulck/sgx-step/tree/master/app/foreshadow 2020.

[114] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium* (2018).

[115] Van Bulck, J., Moghimi, D., Schwarz, M., Lipp, M., Minkin, M., Genkin, D., Yuval, Y., Sunar, B., Gruss, D., and Piessens, F. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P* (2020).

[116] Van Bulck, J., Oswald, D., Marin, E., Aldoseri, A., Garcia, F., and Piessens, F. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *CCS* (2019).

[117] Van Bulck, J., Piessens, F., and Strackx, R. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Workshop on System Software for Trusted Execution* (2017).

[118] Van Bulck, J., Piessens, F., and Strackx, R. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS* (2018).

[119] Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., and Strackx, R. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium* (2017).

[120] van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., and Giuffrida, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS* (2016).

[121] van Kesteren, A. Safely reviving shared memory, https://hacks.mozilla.org/2020/07/safely-reviving-shared-memory/ 2020.

[122] Van Schaik, S., Giuffrida, C., Bos, H., and Razavi, K. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium* (2018).

[123] van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. RIDL: Rogue In-flight Data Load. In *S&P* (2019).

[124] Varadarajan, V., Kooburat, T., Farley, B., Ristenpart, T., and Swift, M. M. Resource-Freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *CCS* (2012).

[125] Vila, P., Ganty, P., Guarnieri, M., and Köpf, B. CacheQuery: Learning Replacement Policies from Hardware Caches. In *PLDI* (2020).

[126] Vila, P., and Köpf, B. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *USENIX Security Symposium* (2017).

[127] Vila, P., Köpf, B., and Morales, J. Theory and Practice of Finding Eviction Sets. In *S&P* (2019).

[128] Votipka, D., Rabin, S., Micinski, K., Foster, J. S., and Mazurek, M. L. An Observational Investigation of Reverse Engineers' Processes. In *USENIX Security Symposium* (2020).

[129] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., and Gunter, C. A. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS* (2017).

[130] Weber, D., Ibrahim, A., Nemati, H., Schwarz, M., and Rossow, C. Osiris: Automated Discovery Of Microarchitectural Side Channels. In *USENIX Security Symposium* (2021).

[131] Weiser, S., Zankl, A., Spreitzer, R., Miller, K., Mangard, S., and Sigl, G. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *USENIX Security Symposium* (2018).

[132] Wichelmann, J., Moghimi, A., Eisenbarth, T., and Sunar, B. MicroWalk: A Framework for Finding Side Channels in Binaries. In *ACSAC* (2018).

[133] Wray, J. C. An Analysis of Covert Timing Channels. *Journal of Computer Security* (1992).

[134] Xiao, Y., Li, M., Chen, S., and Zhang, Y. Stacco: Differentially Analyzing Side-channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *CCS* (2017).

[135] Xiao, Y., Zhang, Y., and Teodorescu, R. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS* (2020).

[136] Xu, Y., Cui, W., and Peinado, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P* (2015).

[137] Yarom, Y. Mastik: A Micro-Architectural Side-Channel Toolkit, https://cs.adelaide.edu.au/~yval/Mastik/ 2016.

[138] Yarom, Y., and Falkner, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).

[139] Yosifovich, P., Ionescu, A., Russinovich, M. E., and Solomon, D. A. *Windows Internals Part 1*, 7 ed. Microsoft Press, 2017.

## A *libtea* Feature List

- **Common**: make architectural and speculative memory accesses; flush and prefetch cache lines; deploy high-resolution timers; handle or suppress exceptions; pin to a core and start victim threads; get physical addresses; read-/write model-specific or system registers; disable hardware prefetchers (Intel only).
- **Cache**: Flush+Reload, Prime+Probe, and Evict+Reload; threshold calibration and eviction set generation; retrieve cache set and slice information; encode/decode data in the cache covert channel; print covert channel histograms.
- **Paging**: print PTEs in a human-readable format; read-/write PTEs and physical pages; map physical address ranges; invalidate TLB entries; read/write memory types (PATs/MAIRs) e.g., to make a region uncacheable; force memory deduplication (Windows only).
- **Interrupts**: print GDT/IDT entries, call gate descriptors, and segment descriptors; establish user-space mappings for GDT/IDT; install a custom call gate; install user mode and kernel mode interrupt handlers; run arbitrary C functions in kernel mode; configure the local APIC timer (for precise execution control).
- **Enclave** *(Intel SGX only)*: register a custom Asynchronous Exit Pointer (AEP) trampoline; get addresses of the enclave base and register state (GPRSGX) region; print/retrieve enclave information and GPRSGX state; read/write enclave addresses (debug enclaves only); fetch the stored instruction pointer (ERIP) from an interrupted enclave.