# PRIDWEN: Universally Hardening SGX Programs via Load-Time Synthesis

Fan Sang*[,1], Ming-Wei Shih*[,3], Sangho Lee[4], Xiaokuan Zhang[1],
Michael Steiner[2], Mona Vij[2] and Taesoo Kim[1]
[1]*Georgia Institute of Technology,* [2]*Intel Labs,* [3]*Microsoft,* [4]*Microsoft Research*

## Abstract

A growing class of threats to Intel Software Guard Extensions (SGX) is Side-Channel Attacks (SCAs). As a response, numerous countermeasures have been proposed. However, it is hard to incorporate them to protect SGX programs against multiple SCAs simultaneously. A naïve combination of distinct countermeasures does not work in practice because some of them are 1) *undeployable* in target environments lacking dependent hardware features, 2) *redundant* if there are already defenses with similar functionalities, and 3) *incompatible* with each other by design or implementation. Identifying all of such conditions and preparing potential workarounds before deployment are challenging, primarily when an SGX program targets multiple platforms that abstract or manipulate their configurations.

PRIDWEN is a framework that selectively applies essential SCA countermeasures when loading an SGX program based on the configurations of the target execution platform. PRIDWEN allows a developer to deploy a program in the form of WebAssembly (Wasm). Upon receiving a Wasm binary, PRIDWEN probes the current hardware configuration, synthesizes a program (i.e., a native binary) with an optimal set of countermeasures, and validates the final binary. PRIDWEN supports both software-only and hardware-assisted countermeasures, and our evaluations show PRIDWEN efficiently, faithfully synthesizes multiple benchmark programs and real-world applications while securing them against multiple SCAs.

## 1 Introduction

Conducting *confidential or private computing* in a shared computing environment (e.g., the public cloud) is challenging [1, 2]. Intel Software Guard Extensions (SGX) [3, 4] has thus been proposed and adopted by leading cloud service providers to help ensure even system software and hardware cannot compromise the authenticity, confidentiality, and integrity of applications running inside SGX *enclaves* [5–7]. Nevertheless, Intel SGX is susceptible to Side-Channel Attacks (SCAs) [8], which are threats to shared cloud environments in which it aims to be deployed. Researchers have shown that SGX is vulnerable to various SCAs utilizing cache [9–13], page table [14–18], and transient execution [19, 20], which can infer sensitive control flows or exfiltrate secret data. To defend against individual SCAs, software- and/or hardware-based countermeasures have been proposed, such as cache flushing [21, 22] or eviction detection [23], page fault detection [24], and Hyper-Threading Technology (HT) disabling [21, 22] or co-location detection [25, 26] (Table 1).

| Attack | Known Countermeasures |
|---|---|
| Cache | Cache flushing [21], cache eviction detection [23] |
| Page | Page fault detection [24], frequent exception monitoring [26, 27] |
| HT | HT disabling [21], co-location detection [25, 26] |
| Branch prediction | Branch obfuscation [28] |
| Speculation | Branch prediction control [29], lfence [30] |
| L1TF | Cache flushing and HT disabling [21] |
| MDS | HT disabling [22] |

**Table 1:** Known side-channel attacks against SGX and countermeasures. HT: Hyper-Threading; L1TF: L1 Terminal Fault; MDS: Microarchitectural Data Sampling.

However, multiple side channels can co-exist in a vulnerable program; protecting SGX programs from multiple known SCAs is difficult, not to mention the existence of unknown ones. One way is collectively applying existing countermeasures against individual SCAs, but naïvely doing so fails due to the unawareness of diverse target execution platforms or co-existing mitigation techniques, which may make such countermeasures 1) *undeployable* due to different hardware settings, 2) *redundant* because of over-protection, and 3) *incompatible* due to conflicts among different mitigations. Another way is adopting a comprehensive countermeasure, i.e., oblivious execution [18, 31], that is effective to many SCAs except for speculative execution. However, even the state-of-the-art oblivious execution incurs average slowdown of 51× [31], largely downgrading the cost-effectiveness of cloud computing. A practical alternative, data-location (re-)randomization [32], incurs relatively small slowdown (8×), but it is still heavy and does not cover control-flow leakage.

One conventional approach to solve such issues is to create a *bloated* application incorporating independently compiled object files for each architecture and runtime detection code, to selectively activate them according to different hardware configurations [33]. This approach, however, is not suitable for Intel SGX: First, checking hardware configurations is sup-

*The two lead authors contributed equally to this work.

ported by system software outside the Trusted Computing Base (TCB); malicious system software can provide misinformation about hardware configurations to SGX applications. Second, the secure memory that enclaves share, Processor Reserved Memory (PRM), is limited [4]; bloated SGX applications can easily occupy a huge portion of it.

**PRIDWEN.** To practically protect SGX programs from various SCAs, we argue that the decision of the SCA mitigations to be applied should be delayed as close to the final execution as possible to best fit the target SGX platform as well as co-existing mitigation techniques. Therefore, instead of adopting the static compilation approach, in this paper, we propose PRIDWEN, a framework that uses *load-time synthesis* to dynamically harden SGX programs by selectively applying different mitigation techniques according to the configurations on the target execution platform. While ensuring the security, PRIDWEN maintains the cost-effectiveness of cloud computing by minimizing the extra effort required for preparation on the tenant side, and the runtime overhead of program synthesis on the cloud side.

PRIDWEN has a universal loader that securely loads and hardens a given SGX program inside an enclave based on four components: 1) Prober that identifies the target platform's hardware and system configurations using SGX exception handling logic and remote attestation; 2) PassManager that manages and determines an optimal set of feasible instrumentation passes based on the identified configurations; 3) Synthesizer that hardens a given SGX program with the chosen instrumentation passes before loading it in the target enclave; and 4) Validator that verifies whether the final executable is hardened as expected, and provides a functionality for developers to remotely verify both the process of synthesis and the hardened binary before execution.

To make PRIDWEN 1) *platform-independent*, 2) *instrumentation-friendly*, and 3) *lightweight*, we develop a new instrumentation framework using WebAssembly (Wasm) [34, 35] as the Intermediate Representation (IR). The size of PRIDWEN in binary is only 1.26 MiB, which only adds a slim footprint to the enclave TCB. Existing Wasm runtimes for SGX [36–39] only *interpret* Wasm binaries without any *instrumentation support*. Furthermore, unlike existing Wasm instrumentation frameworks for non-SGX programs [40, 41], PRIDWEN can comprehensively transform Wasm binaries at both Wasm IR and native code levels. PRIDWEN also supports multiple high-level languages; users only need to compile their SGX programs once with a Wasm compiler backend (e.g., Emscripten [42]).

To showcase the capability and practicality of PRIDWEN, we integrate four mitigation passes into PRIDWEN: 1) T-SGX [24] to prevent a page-fault SCA with a hardware support; 2) Varys [26] to mitigate cache-timing, page-fault-, and HT-based attacks in a software-only manner; 3) QSpectre [30] to mitigate the Spectre attack; and 4) fine-grained Address Space Layout Randomization (ASLR) [43] as a general-purpose mitigation. We first detail the steps to integrate the four passes into PRIDWEN, then we demonstrate how PRIDWEN produces the optimal set of passes based on the runtime configurations with minimal manual effort.

**Performance.** PRIDWEN poses moderate performance overhead on top of the original mitigation techniques and retains faithfulness of execution semantics (§6). The average slowdown of hardened real-world applications (Lighttpd, libjpeg, and SQLite) was 2.1× with hardware-assisted non-redundant mitigation techniques and 3.6× with software-only mitigation techniques for outdated microcode (i.e., no hardware-level mitigation), which closely conforms to the originally reported performance overhead of the selected countermeasures. Also, PRIDWEN faithfully compiled and ran all 73 programs from the official Wasm specification test suite [44]. Program synthesis completed within 0.5 s with a temporary usage of less than 25 MiB of enclave memory across tests.

PRIDWEN is designed to be an easily-extensible universal framework that respects the diversity of computing platforms. PRIDWEN is publicly available as an open-source project [1], allowing communities to test, use, and contribute. We envision that the growing PRIDWEN framework should serve as a hub for the SCA-resistant SGX ecosystem, and potentially other mitigations as well.

**Contributions.** This paper makes the following contributions:

- **The first platform-aware load-time synthesis framework for SGX programs.** To the best of our knowledge, PRIDWEN is the first framework that dynamically synthesizes and hardens SGX programs by applying optimal hardware- and/or software-based mitigations according to the target platform.
- **Attestable in-enclave Wasm instrumentation and compilation toolchain.** A comprehensive instrumentation and compilation toolchain based on Wasm is implemented inside the SGX enclave to enable dynamic program synthesis with attestation. PRIDWEN can instrument Wasm both at IR and native level.
- **Extensive evaluation.** We study the performance of PRIDWEN extensively using benchmarks and real-world applications. The results suggest that PRIDWEN only introduces moderate runtime overhead while preserving the execution semantics.

## 2 Background and Related Work

In this section, we present the background and related work. Additional related work can be found at Appendix A.

**Intel SGX.** Intel SGX is a hardware-based Trusted Execution Environment (TEE) that securely runs a userspace application in an untrusted remote environment, such as the public cloud. Through remote attestation [45], Intel SGX allows a user to

[1] https://github.com/sslab-gatech/Pridwen

load his/her signed program into a remote environment, while helping ensure that the program binary has never been altered. To help secure the code and data of SGX programs, Intel SGX provides an enclave to each program, which is a dedicated secure region of the main memory. The enclave is isolated from any other software including an OS. The code and data stored in the enclave are always encrypted by the Memory Encryption Engine (MEE), and decrypted only when they are loaded into a CPU package (i.e., the cache), to help prevent physical attacks such as a cold boot attack [46].

**Exceptions in SGX.** Conventionally, exceptions are delivered to system software such as OS for further investigation. However, Intel SGX cannot adopt traditional exception handling because system software is untrusted. Instead, Intel SGX defines two mechanisms, Asynchronous Enclave Exit (AEX) and Custom Exception Handler (CEH) [47, 48]. Whenever an exception is generated during an enclave execution, the CPU exits from the enclave asynchronously. During AEX, the original exception number and register context are stored into the State Save Area (SSA) inside the enclave and then overwritten by synthetic data. Further, SGX allows developers to define CEHs to process exceptions inside an enclave; These CEHs can retrieve the SSA to check the stored exception number (`GPRSGX.EXITINFO.VECTOR`) and registers (`GPRSGX.<registers>`), and update them (e.g., `GPRSGX.RIP`) to resume the execution.

**SGX side channels.** SCAs against SGX have been actively studied. Traditional cache SCAs work against Intel SGX with different configurations [9–13, 49]. Recent studies show that page access patterns [14–18], interrupt execution time [50, 51], branch prediction behaviors [28, 52], speculative execution [19, 20] and Intel's internal buffers [53–55] can all be used to infer sensitive information inside enclaves. In response, countermeasures that cope with the fundamental characteristics of the SCAs have been proposed. T-SGX [24] and Cloak [23] use Transactional Synchronization Extensions (TSX) to accurately recognize page faults and cache evictions inside an enclave, respectively. Varys [26] and Déjà Vu [27] aim to detect frequent interrupts or AEXs. Also, since HT enables concurrent SCAs without interrupts, Varys [26] and HyperRace [25] try to prevent an SGX hyperthread from being co-located with other hyperthreads. Disabling HT and/or flushing the L1 cache are also necessary to mitigate recent speculative or transient SCAs [21, 22]. In addition, SGX-LAPD [56] leverages a huge page to degrade the accuracy of the page-level SCA. Lastly, oblivious code execution and data access techniques for Intel SGX [18, 31, 57–60] have been proposed as a general countermeasure against SCAs, but they incur overly high performance overhead. The state-of-the-art ORAM-based system Klotski [61] improves the performance significantly. However, it only defeats controlled-channel attacks [14, 18]. In contrast, PRIDWEN focuses on universally hardening SGX applications by automatically and selectively applying multiple defenses against different SCAs together.

**WebAssembly (Wasm).** The World Wide Web Consortium (W3C) proposes Wasm [34] as a platform-independent compilation target for various high-level languages (e.g., `C/C++` and `Rust`). A Wasm binary has language-like syntax and structure that are suitable for compilation and instrumentation. The basic executable unit of code in Wasm is a module that consists of multiple sections, where each section contains specific definitions of the module such as global variables, functions and a sequence of instructions of each function. Wasm instructions execute on a stack machine, and Wasm supports only the structured control flow such as `if-else` and `loop` without `goto` statements, enabling single-pass fast compilation.

**Memory safety in Wasm.** Wasm maintains a linear memory with a configurable size dedicated to all the memory accesses except for local and global variables. The linear memory is disjoint from other memory regions such as the `code` section and the call stack. As a result, given a buggy Wasm program (e.g., originating from a `C` program with a memory corruption bug), an attacker can only interfere with the data in the linear memory, but cannot tamper with its control flow.

**PRIDWEN and Wasm.** Because the Wasm binary is well-structured and friendly for efficient Just-In-Time (JIT) compilation, PRIDWEN adopts Wasm as IR for supporting load-time synthesis. PRIDWEN also benefits from Wasm's memory-safety feature, mitigating code-reuse attacks against SGX [62, 63]. Moreover, the minimal footprint of Wasm fits PRIDWEN's demand for a compact yet flexible instrumentation and compilation toolchain inside an enclave. Although existing compilers (e.g., LLVM) can fit into enclaves with extended size in new scalable CPUs [64], the TCB size will largely increase, and the migration effort would be significant as well.

## 3 Overview

**Scenario.** In this paper, we consider the widely-used *confidential-computing* scenario on the cloud, where the user wants to utilize the Intel SGX on the cloud to protect his/her data and applications. In this scenario, there are two entities: the cloud and the user. The user runs his/her applications inside enclaves, and wants to protect his/her data against side-channel attacks using PRIDWEN.

**Threat model.** Our threat model is similar to the threat models in other SGX-related studies [14, 24, 26]. Our TCB consists of an SGX enclave provided by an Intel CPU and everything inside the enclave, including PRIDWEN, a target Wasm binary prepared by the user, and the PRIDWEN pass selection policy. We assume that the user uses remote attestation to verify the validity of the CPU and PRIDWEN, and establish a secure channel with PRIDWEN to securely transmit his/her binaries. We assume that adversaries have already compromised the underlying privileged system software (e.g., OS) to attack PRIDWEN and the target binary. Any threats due to poten-

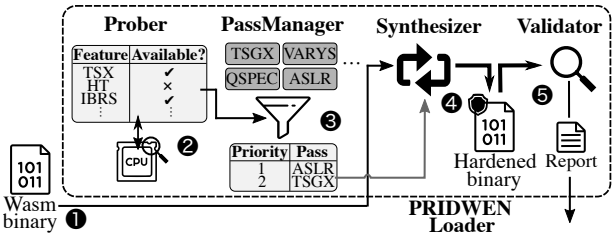


**Figure 1:** Overview of PRIDWEN. ❶ A user compiles a program into a Wasm binary and transmits it to PRIDWEN via a secure channel. ❷ PRIDWEN probes the hardware configurations. In this example, the CPU enables TSX and IBRS while disabling HT. ❸ PRIDWEN selects mitigation passes. Here, it chooses T-SGX and ASLR because the CPU enables TSX (for mitigating page-fault attacks) and IBRS (for mitigating Spectre variants). ❹ PRIDWEN synthesizes and hardens a native binary based on chosen passes. ❺ PRIDWEN validates the final synthesized native binary. A report is sent back to the user to attest the final binary.

tial vulnerabilities of the CPU and the code running insides enclaves are out of scope.

**Goals.** PRIDWEN is designed to achieve the following goals: *1) Adaptivity.* PRIDWEN selects mitigation techniques that conform to the capabilities of the target execution platform on demand. PRIDWEN needs to optimally combine multiple mitigation techniques without causing conflicts or failures.

*2) Attestability.* The second goal of PRIDWEN is to support the remote attestation of the dynamically generated binary inside SGX; Native SGX only supports attesting static binaries. PRIDWEN should allow users to verify the integrity of the final executable running inside an enclave, as well as obtain the genuine information regarding whether the executable is faithfully generated by PRIDWEN (e.g., the selection and application of the mitigation schemes).

*3) Extensibility.* Another goal of PRIDWEN is to be extensible, so that it can support forthcoming mitigation techniques against SCAs besides existing ones. Moreover, it should support multiple platforms due to the diversity of practical computing platforms. The extensibility of PRIDWEN should also allow for smooth integration of legacy mitigation techniques.

**Architecture.** Figure 1 shows an overview of PRIDWEN. The core of PRIDWEN is an in-enclave loader that implements key ideas with corresponding components: *user-mode hardware probing* (Prober), *optimal pass selection* (PassManager), *load-time program synthesis* (Synthesizer), and *post-synthesis validation & final binary attestation* (Validator). Given that each countermeasure may depend on specific hardware features, Prober interacts with the platform and dynamically determines the availability of these features. Based on the probing results, PassManager determines an optimal set of countermeasures (i.e., instrumentation passes) and finalizes the order of applying them based on user policies. Next, PassManager informs Synthesizer about the final selection. Synthesizer takes a Wasm binary (provided by the user via a secure network channel) as an input and compiles it into a native one. During the compilation, Synthesizer hardens the binary with the optimal pass set provided by PassManager. Validator takes the synthesized binary as an input, and verifies that 1) each countermeasure is correctly enforced, and 2) no conflict exists among the enforced countermeasures. Validator also provides the functionality of attestation on the final binary.

```
#define UD 6 /* Invalid opcode exception */
bool tsx_support = false;
check_tsx_support:
  _xbegin();
    tsx_support = true;
  _xend();
exception_handler:
  if (SSA.GPRSGX.EXITINFO.VECTOR == UD &&
      SSA.GPRSGX.RIP == check_tsx_support) {
    GPRSGX.RIP = skip_tsx_check;
  }
```

**Figure 2:** Exception-based probing code for TSX. If a CPU does not support TSX, there will be a `#UD` exception that needs to be handled by an in-enclave exception handler to proceed execution (i.e., changing `GPRSGX.RIP`).

# 4 PRIDWEN

## 4.1 Prober

The goal of Prober is to identify hardware capabilities of the target execution platform, which is needed by PassManager to determine the optimal set of mitigation schemes to enforce. This *hardware probing* step typically requires interactions with the system software, such as retrieving privileged registers (i.e., Model-Specific Register (MSR) and control registers) and executing the `cpuid` instruction. However, we cannot rely on these approaches because the system software is not trusted in our threat model. SGX provides an attribute field called XSAVE-Feature Request Mask (XFRM) to determine whether some hardware features are enabled at enclave creation, but it only covers a few instructions (e.g., AVX and MPX [48]). To solve this, PRIDWEN leverages exception handling and remote attestation to securely probe hardware configurations while running inside an enclave.

**Exception-based instruction probing.** The instruction probing identifies whether PRIDWEN can use hardware-assisted mitigation techniques relying on specific instructions. A CEH for SGX (§2) can be used to determine whether a target system supports or enables the required instruction. Specifically, the probing code executes the specific instruction demanded (e.g., TSX) inside SGX, and then checks whether it results in a `#UD` exception by inspecting the exception information (Figure 2). If it does—i.e., the target platform does not support the instruction, PRIDWEN adopts a software replacement of the hardware-assisted mitigation if available, or omits it otherwise. The CEH then advances `GPRSGX.RIP` to continue execution.

Attackers can disrupt this type of probing, but it only results in Denial-of-Service (DoS) that they can always trigger without special attacks: 1) Attackers can simply resume the

| API | Hooking point |
|---|---|
| `onFunctionStart(CCTX *c)` | Beginning of a function |
| `onFunctionEnd(CCTX *c)` | End of a function |
| `onControlStart(CCTX *c)` | Beginning of a control statement |
| `onControlEnd(CCTX *c)` | End of a control statement |
| `onInstrStart(CCTX *c)` | Before a IR-level instruction |
| `onInstrEnd(CCTX *c)` | After a IR-level instrcution |
| `onMachineInstrStart(CCTX *c, MI *i)` | Before a native instruction |
| `onMachineInstrEnd(CCTX *c, MI *i)` | After a native instrcution |

**Table 2:** The APIs for the instrumentation. `CCTX`: `CompilerContext`. `MI`: `MachineInstr`. `MCTX`: `MachineContext`. `MB`: `MachineBasicBlock`.

enclave execution right after the `#UD` exception without invoking the CEH. However, `GPRSGX.RIP` still points to the invalid instruction, and it is impossible to manipulate it or the exception number outside the enclave. Therefore, this only incurs repeated `#UD` exceptions. 2) Attackers can selectively enable a specific hardware instruction *only during probing* and disable it during the actual execution. This trick can deceive the probing, but it only introduces `#UD` exceptions during runtime, resulting in another DoS.

**Remote attestation for hardware configuration.** The remote attestation determines whether a target platform is vulnerable to SCAs that utilize certain hardware features. SGX remote attestation allows PRIDWEN to accurately determine several hardware configurations, i.e., HT and Indirect Branch Restricted Speculation (IBRS). If a remote device turns on HT, an attestation verification report will contain `CONFIGURATION_NEEDED` in the `isvEnclaveQuoteStatus` field since API version 3 [65, 66]. PRIDWEN can leverage this information to selectively adopt mitigations for preventing hyperthread co-locations [25, 26]. Also, if a remote device does not install the microcode update for indirect branch control mechanisms, a remote attestation protocol will indicate `GROUP_OUT_OF_DATE` [67]. If users still want to securely run their code on such an outdated device, they can adopt software-based approaches [30] against speculative SCAs. Without learning such hardware information, unnecessary performance overhead might be paid for applying redundant protections. It is worth mentioning that updating microcode and changing HT configuration require system reboot in general; As a result, malicious system software cannot manipulate these hardware configurations during the execution of hardened programs.

## 4.2 PassManager

PassManager is in charge of selecting and integrating multiple mitigation schemes. PassManager provides a set of high-level APIs that allows developers of side-channel mitigation schemes to implement their instrumentation passes and plug them into the PRIDWEN loader. During the load time, PassManager 1) maintains a list of plugged-in passes, 2) determines the optimal set of passes for Synthesizer to execute, and 3) resolves the correct application order of each selected pass to avoid conflicts.

**Pass APIs.** Table 2 lists the high-level APIs for implementing instrumentation passes. For instrumentation, we expose all the hooks as APIs. To reflect the structure of a Wasm module, we classify the IR-level hooks into the granularity of functions, controls, and instructions. Each hook can obtain the information about the hooking IR instruction and the current states of compilation via the `CompilerContext` (`CCTX`) data structure. For the native level, the hook should consult the information of the native instruction via the `MachineInstr` (`MI`) data structure, as the `CompilerContext` data structure does not track such information.

**Pass selection and ordering.** When being plugged into the PRIDWEN loader, each pass is associated with a configuration file that specifies the type of SCA to mitigate, hardware features or other passes that it depends on, and a list of passes incompatible with it. Each pass can also specify its weakly dependent passes, which indicates that the pass depends on these weakly-dependent passes only when they are available. During the initialization phase, PassManager adds all the plugged-in passes into a pass queue. For better flexibility, PRIDWEN also allows a user to customize the pass queue by providing a pass selection policy ($P$), which contains descriptions of all plugged-in passes and their dependencies. We detail our implementation of a pass selection policy in §5.2.

To select the optimal set of passes, PassManager takes the following steps: 1) It consults Prober about the current hardware configuration. 2) It checks the dependency of each pass in the queue, and drops a pass if the required hardware feature is not available. 3) It checks the types of side channels that each active pass mitigates; if PassManager identifies more than one passes targeting the same SCA, it retains the one with the highest priority value specified in $P$. If not specified, it will first assign a priority value to each of them based on several criteria including performance overhead. 4) To determine the application procedure of active passes, PassManager builds a dependency graph of all the passes given the dependencies specified in $P$. Next, PassManager uses the topological order of the graph as the application order. PassManager may drop passes if their strong dependencies are not satisfied or incompatible passes are in the active pass set[2]. If all passes are independent, PassManager uses the order in the pass queue as the application order. Here we assume the graph contains no circular dependencies; otherwise, PRIDWEN will terminate the execution.

## 4.3 Synthesizer

Synthesizer uses *load-time synthesis* to dynamically generate a final binary hardened with the optimal set of mitigation passes (§4.2) for the current hardware configuration, and loads

[2]Note that we did not come across any SCA mitigations that are mutually exclusive. If such cases emerge in the future, programmers can specify this situation in the pass selection policy ($P$) and mark the involved mitigation passes as mutually exclusive; the pass with higher priority will be applied by default. Users may override the policy to choose a custom priority.

the binary into memory for execution. Synthesizer adopts a Wasm binary as the input, and takes three steps, i.e., parsing, compilation, and instrumentation, to achieve this goal. We extend the compilation chain to support both IR- and native-level instrumentation so that it is flexible enough to integrate various types of SCA mitigation schemes with PRIDWEN.

**Parsing.** In the parsing step, Synthesizer performs standard decoding on a Wasm binary and converts it into Wasm IR. During decoding, Synthesizer also validates the format of the binary with several checks (e.g., type checking of functions) to guarantee that the binary follows the specification. Any modification to the binary before parsing can thus easily result in an immediate rejection. For example, inserting an instruction that causes the inconsistency on the stack machine renders the binary invalid.

**Compilation.** To generate the native binary inside the enclave given Wasm IR, Synthesizer performs a single-pass compilation over each function (similar to the baseline compilation of SpiderMonkey [34] and V8 [68]). During the compilation of a function, Synthesizer virtually executes each instruction based on the execution model of the Wasm stack machine and generates the corresponding native code. Synthesizer also keeps track of the metadata about each value (e.g., actual location and data type) on the operand stack to help correctly generate the native code and facilitate type-checking. In addition to the operand stack, Synthesizer maintains a control stack that keeps track of the control flow of the function. Pushing a value to the control stack indicates the function initiates a new control statement (e.g., `block`, `if`, or `loop` instruction), while popping a value from the control stack implies reaching the end of the current statement (e.g., an `end` instruction). The control stack provides sufficient information for Synthesizer to resolve the target of a branch (e.g., a `br` instruction). After finishing the native code generation of all functions, Synthesizer performs relocation. This process patches all the unresolved address values in the native instructions, such as `call` and those for memory accesses.

**Instrumentation.** To support flexible instrumentation, we extend the design of the compilation to provide hooks at both IR- and native-level. For IR-level hooks, we place them both before and after the position that Synthesizer processes an IR instruction to support code insertion, modification, and deletion. For each hook, we provide sufficient information about the corresponding instruction and the states of the compilation at the given point, such as the operands and the control stacks. Since Synthesizer may generate more than one native instruction for a single IR instruction, we provide similar hooks at the native level (i.e., surrounding the generation of native instructions) to support mitigation schemes that require the information about native instructions. To support the insertion or the modification of native instructions that require relocation, we provide the option to mark such instructions with symbols. A symbol refers to a target location that allows Synthesizer to recognize and resolve it during the relocation phase.

**Reproducible synthesis.** Since both the compilation and instrumentation are *deterministic*, Synthesizer has a nice property: the synthesis process is reproducible. This property ensures that given the same Wasm code and hardware configuration, the same version of PRIDWEN loader always generates the same final binary.

## 4.4 Validator

The flexibility of instrumentation indicates that an instrumentation pass can arbitrarily modify the binary. Such modifications can potentially disturb the already applied instrumentation passes or break the binary itself. To avoid such cases, Validator supports *post-synthesis validation* to validate whether the synthesized executable is hardened as expected. Also, since the runtime behavior of PRIDWEN cannot be determined beforehand, Validator provides *final binary attestation* to allow users to remotely verify both the process of synthesis and the hardened binary before execution. Both *post-synthesis validation* and *final binary attestation* are necessary to ensure the correctness of the final binary. In addition, they are conducted *only* once before running the program, and thus will not affect the runtime performance.

**Post-synthesis validation.** In post-synthesis validation, Validator conducts static analysis over a synthesized binary. Unlike typical binary analysis that assumes a stripped binary, post-synthesis validation enables more sophisticated analyses by taking advantage of the metadata (e.g., the control-flow information) provided by Synthesizer. Post-synthesis validation takes in the form of validation passes coupled with each instrumentation pass. Based on the control-flow information, Validator executes validation passes at the basic-block level. Validator iterates through all functions in the binary and invokes a procedure implemented in each validation pass at the beginning of each basic block, which performs a series of checks based on the content of the basic block (i.e., raw bytes). For example, a procedure can determine whether specific instrumentation is applied based on pattern matching. If any of the validation passes fails, Validator rejects the binary. Optionally, the procedure can utilize other metadata such as the original IR instructions that map to the basic block to facilitate the analysis beyond binary scanning.

**Final binary attestation.** In addition to using remote attestation for hardware probing (§4.1), PRIDWEN uses remote attestation to attest the dynamically synthesized binary inside the enclave. SGX does not natively support the attestation of dynamic enclave content; instead, traditional remote attestation measures only the static code and data that are initially inside an enclave, which is used as a piece of evidence throughout the process of remote attestation. To attest dynamic content, PRIDWEN incorporates a two-step scheme that extends the attestation of static content.

| Attack surface | SW-only Mitigation | HW-assisted Mitigation |
|---|---|---|
| **Cache timing** | Interrupt (**Varys**) | Cache flushing (microcode) |
| **Page fault** | Interrupt (**Varys**) | **T-SGX** |
| **HT** | Co-location (**Varys**) | HT disabling (microcode) |
| **Speculative execution** | **QSpectre** | IBRS (microcode) |
| **Static layout** | **ASLR** | N/A |

**Table 3:** Attack surfaces and software-only or hardware-assisted mitigation schemes PRIDWEN implements. CPUs with recent microcode update do not have some of the attack surfaces.

In the first step, a user uses the SGX standard procedure to attest the static part of PRIDWEN and establishes a secure channel [45]. Then, the user sends a Wasm binary `p.wasm` to PRIDWEN via the secure channel and PRIDWEN starts to synthesize the final binary based on the hardware configuration (`hw_config`) of the execution platform. In the second step, PRIDWEN sends the user: 1) the measurement of the synthesized binary `p.code` (i.e., the hash of native code blocks `hash(p.code)`) and 2) the `hw_config`. The user can then validate `hash(p.code)` thanks to a PRIDWEN's property: *reproducible synthesis* (§4.3). With the reproducible synthesis, the user can validate the final binary based on both `p.wasm` and `hw_config`.

# 5 Implementation

We implement a prototype of PRIDWEN with 25k lines of `C` code on top of the Intel Linux SGX SDK 2.5.102. The size of PRIDWEN in binary is only 1.26 MiB, maintaining a slim footprint of trusted computing base (TCB). For native code generation, we implement an `x86` backend to support the full Wasm instruction set.

**Runtime support.** Our prototype provides an Emscripten-compatible runtime support that allows to run fairly large, complex applications such as Lighttpd, as shown in §6. The application is directly compiled from unmodified `C` source code to a Wasm binary using the Emscripten compiler.

**Attestation of synthesized binaries.** Our prototype supports *final binary attestation* mentioned in §4.4. Also, the prototype provides a tool that allows users to locally validate the measurement of the synthesized binary.

## 5.1 Example Passes

To illustrate the capability and practicality of PRIDWEN, our prototype implementation integrates four SCA mitigation schemes (ASLR [43], Varys [26], T-SGX [24], and QSpectre [30]) into PRIDWEN using provided APIs, and *simultaneously* cover five important SCA surfaces (cache timing, page fault, HT, speculative execution, and static layout) (Table 3) based on the probed hardware configuration (§4.1). Although the four mitigation schemes were not originally introduced by this work, they are selected to demonstrate how to integrate existing mitigation techniques using PRIDWEN. Because the control-flow information (including the definition of basic blocks in the binary) is required by most of the passes, we implement a control-flow analysis pass to share the information with other passes, eliminating the overhead posed by repetitive analyses. PRIDWEN helps to prioritize hardware-assisted mitigations with lower overheads if the execution platform supports them (e.g., TSX for T-SGX), and safely avoid redundant countermeasures if the platform is free from the corresponding SCAs thanks to recent microcode or hardware updates [21, 22, 29, 69].

### 5.1.1 Example Pass #1: Fine-grained ASLR

Many SCAs rely on accurate memory layout information to improve the granularity of leaked information. Thus, PRIDWEN enables fine-grained ASLR by default, which randomizes the location of every basic block, as a general mitigation scheme against SCAs.

**Integration.** We adopt the similar compiler-level scheme from SGX-Shield [43] by inserting a `jmp` instruction at the end of every basic block. First, the pass uses the `onControlStart` and `onControlEnd` APIs (Table 2) to identify the structure of a basic block. Next, the pass inserts a `jmp` with a symbol that points to the succeeding basic block if it does not end with a `jmp`. The pass also updates the targets of other branches accordingly by using the `onMachineInstrEnd` API. Later, Synthesizer shuffles the placement of each basic block if the ASLR pass is enabled. During the relocation, the generated symbols allow Synthesizer to resolve the target of each branch to a basic block at a randomized location.

**Post-synthesis validation.** We integrate a validation pass that performs the following checks: 1) whether a basic block terminates with a `jmp` and 2) whether each branch points to the correct target based on the control-flow information.

### 5.1.2 Example Pass #2: T-SGX

SGX allows the OS to handle page faults. Page-fault SCAs [14] exploit this design decision by intentionally making enclave pages inaccessible and observing which pages are accessed. To defeat this SCA, T-SGX [24] hides page faults from the OS by running an enclave spitted as small code blocks inside TSX transactions. As a result, all page faults occurring during the execution are suppressed (i.e., not delivered to the OS).

**Integration.** Our T-SGX pass has cache usage and execution time analyzers for native instructions using the `onMachineInstrEnd` API. Based on the analysis results, the pass determines the scale of a code block. Next, the pass replaces branch instructions at the end of the block with the instructions redirecting to the springboard (i.e., a `lea` for saving the address of the next code block and a `jmp` to the springboard). Similar to the fine-grained ASLR pass, the T-SGX pass identifies basic blocks in the binary by using the `onControlStart` and `onControlEnd` APIs. To support the springboard, the pass places the springboard code before

the entry function (e.g., `main`) of the binary by using the `onFunctionStart` API.

**Post-synthesis validation.** The validation pass for T-SGX checks 1) the presence of the springboard, 2) the presence of the instructions to jump to the springboard at the end of every code block, and 3) whether the target of the instructions in step 2 correctly points to the springboard. Optionally, the pass can re-analyze cache usage and execution time to ensure the correctness of the code splitting.

#### 5.1.3 Example Pass #3: Varys

SCAs usually require frequent interrupts or HT to accurately identify the execution context (e.g., which pages are accessed) or to attack in-core cache and speculation units. Varys [26] is a software-based approach to detect such behaviors.

**High-frequency AEX detection and cache eviction.** Varys identifies the occurrence and frequency of interrupts during the enclave execution by analyzing AEXs. Specifically, whenever an AEX occurs, SGX updates the corresponding field in the SSA. Thus, by counting the number of instructions executed at every basic block and periodically polling the SSA, Varys can estimate the frequency of AEXs. In addition, Varys explicitly evicts cache lines upon detecting AEXs to mitigate cache-based SCAs.

**Co-location test.** To prevent scheduling victim and attack threads to the same HT core, Varys prepares a pair of SGX threads and checks whether they are in the same core via an L1-cache-based covert timing channel. Varys performs this co-location test whenever it observes an AEX.

**Integration.** PRIDWEN's Varys pass inserts the checking code at the beginning of every basic block by using the `onControlStart` and `onControlEnd` APIs. Unlike the original Varys that counts the number the instructions at the LLVM IR level, our pass counts the number of native instructions with the help of the `onMachineInstrEnd` API. For the SSA polling routine, the pass inserts the code before the entry function of the binary via the `onFunctionStart` API. The pass also adds the co-location test code to the SSA polling routine (i.e., after detecting an AEX).

**Post-synthesis validation.** The validation pass verifies 1) the presence of the checking code, 2) the correctness of the instruction number added to the counter, 3) the presence of the SSA polling code, and 4) whether the target of the `call` in the checking code points to the SSA polling routine.

#### 5.1.4 Example Pass #4: QSpectre

One software-based approach to mitigate the Spectre attack is to use serializing instructions (e.g, `lfence`) to prevent the CPU from speculatively executing instructions beyond the intended placements. Following this idea, Microsoft Visual Studio has adopted a compiler-based scheme, QSpectre [30],

```
1  [ { "name": "tsgx",
2      "sca": [ "page" ],
3      "dependency": { "hw": [ "tsx" ], "weak": [ "aslr" ] },
4      "priority": "high"
5    },
6    { "name": "cotest-tsgx",
7      "sca": [ "ht" ],
8      "dependency": { "hw": [ "ht" ], "strong": [ "tsgx" ] },
9      "priority": "high"
10   },
11   { "name": "qspectre",
12     "sca": [ "spectre" ],
13     "dependency": { "hw": [ "!ibrs", "ht" ] },
14     "priority": "high"
15   },
16   { "name": "varys",
17     "sca": [ "cache", "page" ],
18     "dependency": { "hw": [ "!cache" ] },
19     "priority": "medium"
20   },
21   { "name": "cotest-varys",
22     "sca": [ "ht" ],
23     "dependency": { "hw": [ "ht" ], "strong": [ "varys" ] },
24     "priority": "medium"
25   },
26   { "name": "aslr",
27     "priority": "high" } ]
```

**Figure 3:** Example pass selection policy $P$ (a `.json` file). `name`: name of the current pass; `sca`: the SCAs to mitigate; `dependency`: the required hardware (`hw`) and dependent passes (can be `weak` or `strong`); `priority`: the priority of the current pass.

which finds potentially vulnerable code patterns and inserts `lfence` instructions During compilation.

**Integration.** Instead of inserting `lfence` instructions based on pattern matching, which can be bypassed [70], PRIDWEN's instrumentation pass for QSpectre adopts a simple, yet effective strategy: inserting `lfence` instructions to all `if-else` structures. More concretely, the pass inserts an `lfence` instruction right after the conditional branch in the code of an `if-else` structure. This pass uses the `onMachineInstrEnd` API and determines if a conditional branch is in an `if-else` structure by consulting the `CompilerContext` data structure.

**Post-synthesis validation.** The validation pass simply checks the presence of the `lfence` instruction in every `if-else` structure.

### 5.2 Pass Coordination

PassManager selects the optimal set of passes and resolves potential conflicts following the steps mentioned in §4.2. To incorporate the four passes into PRIDWEN, we specify the pass selection policy $P$ (shown in Figure 3). Note that the policy $P$ is just an example; PRIDWEN can transparently support any developer-provided policies by design.

**Pass selection.** First, PassManager selects all feasible passes according to hardware dependencies. For example, it selects the QSpectre pass if IBRS is disabled and HT is enabled (Line 13). Next, it prunes passes that target overlapping SCAs based on the `priority`. We prioritize T-SGX (hardware-based) over Varys (software-based) in $P$ such that if PassManager has selected the T-SGX pass because TSX is available, it will omit the Varys pass. Then, PassManager checks the unprocessed

passes in *P* and includes those whose dependencies are all satisfied (e.g., co-location test for T-SGX `cotest-tsgx`), or without any dependency (e.g., ASLR).

**Pass ordering.** Based on the dependencies specified in *P*, PassManager also determines the order of applying passes to resolve potential conflicts among them. For example, the ASLR and T-SGX passes can compete with each other to instrument branches at the end of basic blocks. To avoid such conflicts, a `weak` dependency between the two passes is indicated in *P* (Line 3). Accordingly, PRIDWEN serves the ASLR pass first and then applies the T-SGX pass with slight modification (e.g., instrument `jmp`s inserted by the ASLR pass to make it point to the T-SGX springboard). Also, PRIDWEN must apply the T-SGX co-location test pass `cotest-tsgx` after the T-SGX pass itself to correctly insert the testing code at the springboard, which is indicated in *P* as a `strong` dependency (Line 8). As passes without dependencies (e.g., QSpectre) can be applied at anytime, PRIDWEN simply applies them after serving passes with dependencies.

# 6 Evaluation

We evaluate PRIDWEN on successful mitigation of individual targeted SCAs (§6.1), the semantic correctness of the input Wasm program (§6.2), the performance characteristics of the PRIDWEN loader (§6.3), and the performance overhead of PRIDWEN-synthesized binaries (§6.4).

**Experiment setup.** We ran all the experiments on a machine with a 4-core Intel i7-6700K CPU (Skylake microarchitecture) operating at 4 GHz with 32 KiB L1 and 256 KiB L2 private caches, an 8 MiB L3 shared cache, and 64 GiB of RAM. The machine was running Linux kernel 4.15. The PRIDWEN loader is compiled with gcc 5.4.0 and executed on top of the Intel Linux SGX SDK 2.5.102.

**Applications and test suites.** We use three real-world applications or libraries (Lighttpd 1.4.48 [71], libjpeg 9a [72], and SQLite 3.21.0 [73]) as a macro-benchmark suite representing large, complex applications, as well as a micro-benchmark suite, PolyBenchC [74]. The benchmark suite consists of 23 small `C` programs with only numerical computations (i.e., no `syscall`) that are used to evaluate the runtime performance of just-in-time compiled Wasm binaries against native C binaries [34]. We compile the original source code of each micro- or macro-benchmark program into Wasm using Emscripten [42], an LLVM-based compiler. We also directly port all of the programs using SGX SDK to serve as baseline versions. We use the official Wasm specification test suite [44] to test the correctness of the synthesis of PRIDWEN (§6.2).

**Methodology.** For each run of experiments, we take the compiled Wasm binaries as input to PRIDWEN. To evaluate PRIDWEN-synthesized binaries with distinct sets of defense schemes enforced, we manually configure PRIDWEN before each run. We use **BASE** to represent the configuration of baseline compilation (i.e., synthesis without instrumentation) and the name of defense schemes to represent the configuration of enforcing the corresponding schemes. For example, **TSGX** indicates the configuration with T-SGX enforced. For the ease of comparing Varys and T-SGX, the rest of the section uses **VARYS**, to represent its original design with the co-location test, respectively. **QSpectre** or **QS** represent QSpectre. To measure the execution time of each application, we use the `rdtsc` instruction via an `OCall` inside an enclave. The reported results are averaged over 10 runs.

## 6.1 Security Analysis

In addition to statically checking the enforcement of each mitigation scheme via validation passes, we also manually verify whether the integrated versions of example passes are effective and compatible by running simplified SCAs against them and building a test suite (§6.2). After hardening a test binary over the SCA surfaces with different combinations, we introduce frequent page faults and interrupts, manipulate processor affinity, and run a simple Spectre attack [19]. We confirm that the T-SGX pass suppresses page faults during runtime, the Varys pass detects frequent interrupts and thread co-location (if HT is enabled), and the QSpectre pass disrupts speculation (if IBRS is disabled). In addition, combining the ASLR pass and frequent interrupt detection (the Varys pass) can effectively mitigate or slow down an attacker's attempts to infer the fine-grained memory layout.

## 6.2 Correctness

To validate whether the synthesized program behaves as expected, we use the official Wasm specification test suite [44], which provides comprehensive test cases for all Wasm instructions. The test suite consists of 73 programs. Each program includes a set of functions and test cases that specify the expected outputs with given inputs. We ran the test suite on PRIDWEN with all hardening configurations and reported the results in terms of *pass* or *fail* on each program. In addition to the test suite, we also record intermediate values of all benchmark programs (by manually inserting `printf`) for both baseline and PRIDWEN-synthesized version and compare them.

**Results.** The results show that programs with all hardening configurations successfully pass all the test cases, which indicates that 1) the baseline compilation of PRIDWEN (**BASE**) faithfully follows the specification of Wasm, and 2) the enforcement of schemes does not modify the behavior of the program. Moreover, there is no difference when comparing intermediate values between **BASE** and the synthesized binaries.

## 6.3 Performance of PRIDWEN

To show both runtime and memory overheads of the PRIDWEN loader, we measured the execution time that PRIDWEN takes to generate native C binaries and the additional memory that it allocates during the entire process (by

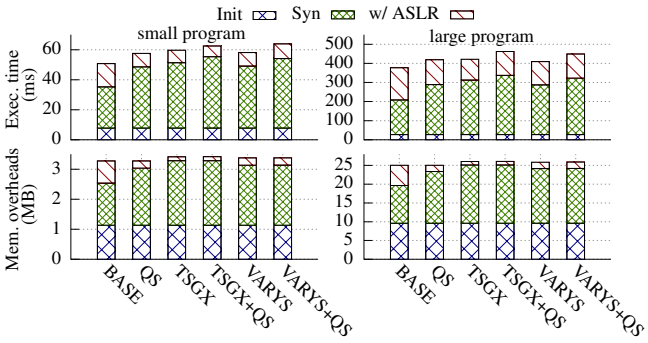


**Figure 4:** The top and bottom figures show runtime overheads and memory overheads, respectively, of PRIDWEN on program synthesis.

hooking `malloc`). To demonstrate the impact on the size of input, we used one small (`2mm`, 52 kB) and one large (`lighttpd`, 462 kB) Wasm binaries as inputs. We also ran experiments with different configurations of PRIDWEN to show the impact of enforcing different mitigations. As the co-location test depends on either T-SGX or Varys and requires only adding a piece of code to each scheme, we do not include it the test in the selected configurations. Note that the overhead only needs to be paid once during the first initialization and synthesis.

The results are shown in Figure 4. We divide each bar into three parts: the initialization stage (blue), the synthesis stage (green), and additional overhead when the ASLR is enforced on top of the corresponding configuration (red). The initialization stage includes the time spent on hardware probing, PassManager initialization, and Wasm parsing. The synthesis stage represents the time spent on compilation and instrumentation in Synthesizer.

**Runtime performance.** For the runtime overhead (Figure 4), it is clear that given the same program, the execution time of the initialization stage is fixed regardless of the configurations. For the large program, PRIDWEN spends more time during the initialization stage, which is mostly due to the process of parsing the Wasm binary; however, the proportion of the execution time spent in the initialization stage decreases, which indicates that PRIDWEN spends more time on the synthesis stage for the large program. Also, enabling ASLR for the large program incurs higher overhead since it has more basic blocks. In addition, enforcing more schemes incurs higher overheads as expected. Overall, the one-time overhead of PRIDWEN is acceptable (less than 500 ms for the large program).

**Memory overhead.** For the memory overhead (Figure 4), the results show that PRIDWEN requires a fixed amount of memory during the initialization stage for the same program. PRIDWEN requires more memory for the large program, since the majority of the required memory is used to store the IR of the input program during the parsing process. We also observe similar memory requirements for PRIDWEN with the **BASE** configuration in the synthesis stage, since it needs more memory to maintain the metadata during compilation for the large program. Also, enabling ASLR on top of **BASE** incurs

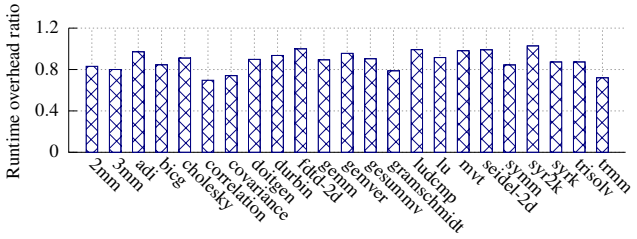


**Figure 5:** The runtime performance of PRIDWEN-synthesized Wasm programs compared to native C binaries.

the highest overhead. The reason is that the instrumentation passes of each scheme all depend on the pass that manages the control-flow graph (CFG) information. As the ASLR pass can share the CFG information with other passes and can directly reuse such information during runtime, enabling ASLR on top of them incurs less overhead; when enabling just ASLR in **BASE**, it needs to generate the CFG information itself, resulting in a large memory overhead. Note that the memory overhead is only imposed once before the execution of the synthesized binary; thus it does not affect the memory usage of the binary in run-time.

## 6.4 Performance of Synthesized Binaries

We measure the runtime and memory overheads of PRIDWEN-synthesized binaries. We compare the results with those of native C binaries ported directly into SGX enclaves. In addition, we measure the performance overhead of the PRIDWEN-synthesized version of the defense schemes by comparing the results with those of the **BASE** configuration, and match it to that of the original implementations (i.e., the overhead indicated in the original papers). We confirm that the overheads are mainly inherited from the original design of the countermeasures; PRIDWEN only imposes minimal amount of overheads in the binaries.

**Runtime performance.** Figure 5 shows the results of running the PolybenchC with the **BASE** configuration, which are normalized to the execution time of the native C programs. Our results indicate that PRIDWEN-synthesized binaries have negligible slowdown or are even faster than the native C binaries, without any mitigation schemes enforced. The execution time of PRIDWEN-synthesized binaries are 0.7×–1.0× of that of the native C binaries. Likewise, the very initial evaluation [34] of the in-browser Wasm compiler reports similar execution overhead results on PolybenchC programs (0.5×–1.4× of that of native C). The runtime performance of PRIDWEN-synthesized Wasm programs is comparable to or even better than that of C programs. This might be due to the small size of PolybenchC programs, with only numerical computations. Therefore, the synthesized Wasm programs are fairly compact and similar to native binaries. Furthermore, the difference between compilers (i.e., Emscripten and GCC) may also contribute to the results. When programs get more complex, performance overhead increases as their Wasm forms

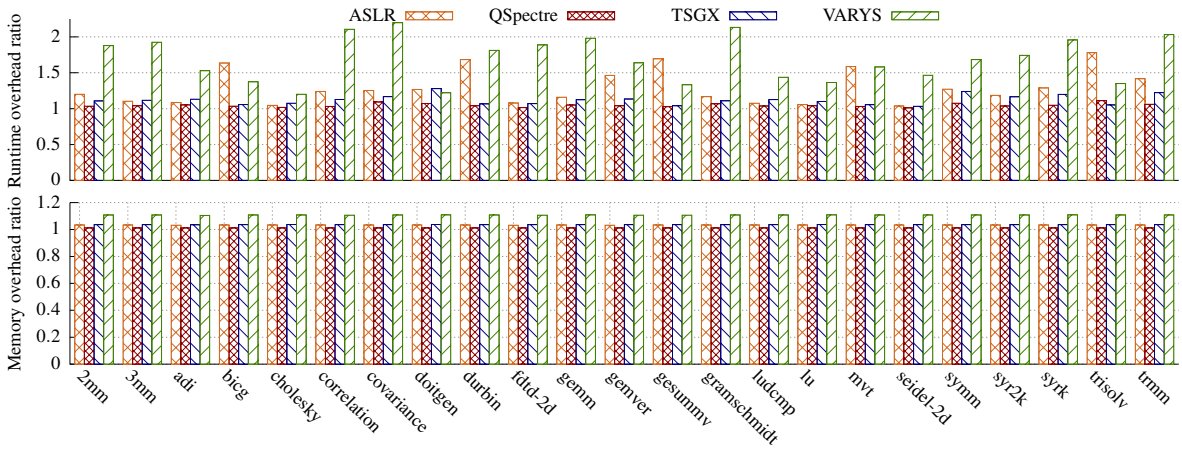


**Figure 6:** The top and bottom figures show the runtime performance and memory overheads, respectively, of PRIDWEN-synthesized programs secured with different mitigation schemes, compared to **BASE**.

no longer maintain the similarity to native binaries (e.g., the number of instructions with one-to-many mappings grows).

Figure 6 demonstrates the results of running PolybenchC with defense schemes enforced. The bar on the figure represents the relative execution time of the program to the **BASE** configuration. **ASLR** incurs various overheads due to different numbers of randomized basic blocks being executed, which is not cache-friendly. Similarly, **QSpectre** also incurs various but smaller overheads, which result from the number of `lfence` instructions being executed.

Regarding T-SGX and VARYS, **TSGX** incurs less overhead than **VARYS** does. Especially, **VARYS** suffers from high overhead when it needs to check AEXs inside a loop structure (see an example in Appendix B). **VARYS** cannot avoid this issue without compromising its security guarantees. In contrast, **TSGX** supports loop optimization, which puts an entire loop into single transaction when possible.

**Memory overhead.** Figure 6 shows how much memory each mitigation demands on top of the binaries with **BASE**. On average, the memory overheads of the synthesized binaries are about 1.2× compared to the baseline binaries, which is moderate.

**Real-world applications.** We use three real-world applications as case studies to show that PRIDWEN provides sufficient support for large, complex programs. In addition to the *Lighttpd* (a web server), the other two applications are based on *libjpeg* and *SQLite* libraries. The *libjpeg* application supports both compressing and decompressing a *jpeg* image and the *SQLite* application supports basic database operations, including `insert`, `select`, `update`, and `delete`. We use the HTTP benchmarking tool, *wrk*, for evaluating the throughputs of the *Lighttpd*. For the other two applications, we measure the execution time of each supported operation and report the average values over 10 runs.

Figure 7 shows the results of *Lighttpd*. The slowdown of **BASE** is 1.5× to the native version. **TSGX** incurs less overheads compared to **VARYS** (1.9× versus 2.6×), demonstrating the advantage of hardware-assisted mitigation schemes over software-only ones. **ASLR** incurs significant overhead because *Lighttpd* has a large number of small-sized basic blocks; This shortens the gap between **TSGX** and **VARYS** when they are enforced with **ASLR**. When enforcing multiple mitigation schemes, the slowdown of *Lighttpd* is up to 4.6× compared to the native binary.

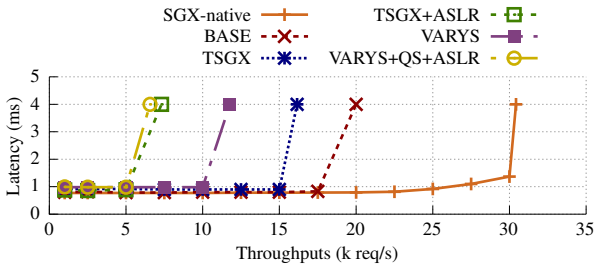


**Figure 7:** The performance of *Lighttpd*; QS: QSpectre.

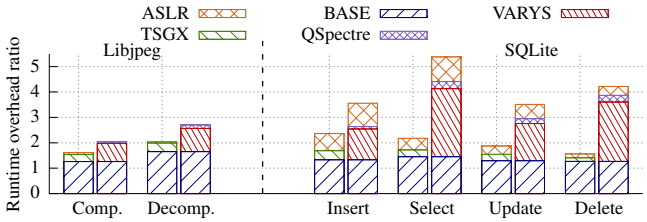


**Figure 8:** The performance of synthesized libjpeg and SQLite; each left bar illustrates hardware-assisted passes, while each right bar illustrates software-only passes.

Figure 8 presents the performance of *libjpeg* and *SQLite* applications. We use stacked bars to represent the incurred overheads when applying the optimal set of mitigations on top of a viable hardware configuration. The runtime overhead of **BASE** is 1.2×–1.7× compared to the native versions. The overheads of individual mitigation schemes are similar to the results of PolyBenchC presented in Figure 6 (e.g., **TSGX** in-

curs less overheads than **VARYS**). The average slowdown of hardware-assisted mitigation schemes is 1.9× while that of software-only mitigation schemes is 3.4×. Depending on hardware configurations, hardware-assisted mitigation schemes are 2.1×–5.4× faster than software-only ones.

Again, as PRIDWEN only aims to integrate multiple mitigation techniques, most of the performance overheads are inherited from the original design of the countermeasures, while PRIDWEN itself does not impose significant overheads.

## 7 Discussion

**Adding new defenses to PRIDWEN.** PRIDWEN is designed with future scenarios in mind. Thanks to the high-level instrumentation APIs provided by PRIDWEN (Table 2), new instrumentation-based defense techniques can be easily added to PRIDWEN as a pass. PRIDWEN pass APIs offer different instrumentation granularity with sufficient information about the corresponding instruction on both IR- and native-level, which should meet all instrumentation needs. We recommend to develop new defenses directly with PRIDWEN to save the hassle of replacing heterogeneous instrumentation APIs with PRIDWEN versions during integration. The developer will also need to provide the type of side-channel attack targeted by the new defense, and the possible dependency on specific hardware features or existing techniques in the pass selection policy. This allows Prober and PassManager to resolve potential incompatibility issues and conflicts, and correctly enforce the new defense. Compared to the effort needed to write a new pass, writing a pass selection policy should be much simpler.

**Upgrade PRIDWEN.** Modern hardware is evolving quickly with updated features useful for security purposes and PRIDWEN is designed to keep up with the hardware evolution. It is necessary for PRIDWEN to allow probing of new hardware features for defense techniques built with such features. The probing logic for the specific hardware feature will need to be added by PRIDWEN developers using either exception-based instruction probing or trusted remote attestation (§4.1) to bypass untrusted privileged software. Novel and secure techniques are also welcomed to probe the hardware capability of the platform. We hope that PRIDWEN can motivate hardware manufacturers to provide official secure probing helpers for hardware features. Upgrade of PRIDWEN is the effort of both the hardware and the software communities. The power of PRIDWEN is truly unleashed when equipped with the most updated inclusion of hardware features and mitigation schemes.

**The recent SmashEx attack [75].** A recent paper presents the SmashEx attack targeting the SGX SDKs; the targeted SDKs do not properly handle *re-entrancy* in their asynchronous exception handling logic, which allows the attacker to compromise the integrity of the SSA region and modify `GPRSGX.RIP`. PRIDWEN is built on top of the trusted Intel SGX SDK. The vulnerability of asynchronous exception handling in SGX (SmashEx) is rooted in the SGX SDK, and it was already mitigated by recent patches [76, 77]. That is, the enclave-specific SSA that hosts the GPRSGX.RIP cannot be compromised with the attack in the latest SDK. PRIDWEN should work with the latest SGX SDK because it does not heavily rely on or modify a certain SDK version.

**Program synthesis on the cloud side vs. the user side.** PRIDWEN makes the design decision of conducting program synthesis on the cloud side to minimize the extra effort required for preparation on the user side. An alternative solution would be deploying a dedicated program on the cloud to report the hardware configuration back to the user, then the user in return synthesizes and caches the hardened binary themselves and sends the binary to the cloud for deployment. Synthesizing and caching the binaries of different configurations at the user side can save compilation cost on the cloud side; however, this puts more burden on the user. In addition, it would be difficult to update the binaries on the cloud side when the configuration changes, since the server has to wait for the user to compile and transmit the updated version. In contrast, when the configuration on the server is changed (e.g., after reboot), PRIDWEN automatically re-synthesizes the application and applies the change upon restarting. It is also possible to optimize PRIDWEN in a similar way by caching the synthesized programs of different configurations on the cloud side to reduce the compilation overhead.

## 8 Conclusion

PRIDWEN is a framework to dynamically synthesize a secure SGX program that is optimally hardened against various SCAs simultaneously, while preventing any *deployability*, *redundancy*, or *incompatibility* problem. To overcome the restrictions of the static deployment model of SGX, PRIDWEN adopts Wasm as the IR, and supports smooth integrations of instrumentation passes for both hardware-assisted and software-only mitigations. PRIDWEN selects an optimal set of mitigations to be applied at runtime according to the hardware configurations of the target platform, and provides means for the user to validate and attest the final synthesized binary. We implement a prototype of PRIDWEN, which integrates four SCA defenses. Through extensive evaluation, we show that PRIDWEN efficiently hardens SGX programs with chosen defenses, while incurring moderate performance overhead.

## 9 Acknowledgment

We would like to thank the anonymous reviewers and our shepherd for their helpful feedback. We also would like to thank Scott Constable and Yuan Xiao from Intel for constructive discussions. This research was funded by Intel and the NSF award NSF-1563848.

## References


[1] J. Mangalindan, "Is User Data Safe in the Cloud?" http://tech.fortune.cnn.com/2010/09/24/is-user-data-safe-in-the-cloud, September 2010.

[2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Nov. 2009.

[3] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Tel-Aviv, Israel, 2013, pp. 1–8.

[4] Intel, "SGX Tutorial, ISCA 2015," http://sgxisca.weebly.com/, Jun. 2015.

[5] N. Porter, "Introducing Asylo: an open-source framework for confidential computing," 2018, `https://cloud.google.com/blog/products/gcp/introducing-asylo-an-open-source-framework-for-confidential-computing`.

[6] Microsoft, "Open Enclave SDK," 2019, https://openenclave.io/sdk/.

[7] Microsoft Azure, "Azure Confidential Computing," 2019, https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[8] S. Johnson, "Intel SGX and Side-Channels," https://software.intel.com/en-us/articles/intel-sgx-and-side-channels.

[9] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, BC, Canada, Aug. 2017.

[10] M. Hähnel, W. Cui, and M. Peinado, "High-Resolution Side Channels for Untrusted Operating Systems," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

[11] F. Dall, G. D. Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks," in *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2018.

[12] A. Moghimi, T. Eisenbarth, and B. Sunar, "MemJam: A false dependency attack against constant-time crypto implementations in SGX," in *Cryptographers' Track at the RSA Conference*. Springer, 2018.

[13] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct.–Nov. 2016.

[14] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[15] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[16] S. Weiser, R. Spreitzer, and L. Bodner, "Single Trace Attack Against RSA Key Generation in Intel SGX SSL," in *Proceedings of the 13th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Seoul, South Korea, Jun. 2018.

[17] J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx, "Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2018, pp. 44–60.

[18] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing Your Faults From Telling Your Secrets," in *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May–Jun. 2016.

[19] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[20] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT)*, Baltimore, MD, Aug. 2018.

[21] Intel, "Q3 2018 Speculative Execution Side Channel Update," 2018, https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html.

[22] ——, "Intel Side Channel Vulnerability MDS," 2019, https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html.

[23] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[24] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[25] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[26] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jul. 2018.

[27] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with Déjá Vu," in *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Abu Dhabi, UAE, Apr. 2017.

[28] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[29] Intel, "Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088," 2018, https://software.intel.com/security-software-guidance/software-guidance/branch-target-injection.

[30] A. Pardoe, "Spectre mitigations in MSVC," 2018, https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/.

[31] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A Commodity Obfuscation Engine on Intel SGX," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[32] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2019.

[33] GCC team, "Using the GNU Compiler Collection (GCC): x86 Built-in Functions," 2019, https://gcc.gnu.org/onlinedocs/gcc/x86-Built-in-Functions.html.

[34] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to Speed with WebAssembly," in *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Barcelona, Spain, Jun. 2017.

[35] WebAssembly Community Group, "WebAssembly Specification: Release 1.0," Tech. Rep., May 2019.

[36] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards Memory Safe Enclave Programming with Rust-SGX," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[37] W. Qiang, Z. Dong, and H. Jin, "Se-Lambda: Securing Privacy-Sensitive Serverless Applications Using SGX Enclave," in *International Conference on Security and Privacy in Communication Systems (SecureComm)*, 2018.

[38] Red Hat, "Enarx," 2019, https://enarx.io.

[39] Intel, "WebAssembly Micro Runtime," 2019, https://github.com/bytecodealliance/wasm-micro-runtime.

[40] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "SEISMIC: SEcure in-lined script monitors for interrupting cryptojacks," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 122–142.

[41] D. Lehmann and M. Pradel, "Wasabi: A Framework for Dynamically Analyzing WebAssembly," in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, Apr. 2019.

[42] "emscripten," 2015, https://emscripten.org/.

[43] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[44] "Mirror of the spec testsuite," 2019, https://github.com/WebAssembly/testsuite.

[45] Intel, "Code Sample: Intel Software Guard Extensions Remote Attestation End-to-End Example," 2018, https://software.intel.com/en-us/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example.

[46] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[47] Intel, "Exception Handling in Intel Software Guard Extensions (Intel SGX) Applications," 2019.

[48] ——, "Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4," May 2019.

[49] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[50] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[51] W. He, W. Zhang, S. Das, and Y. Liu, "Sgxlinger: A new side-channel attack vector based on interrupt latency against enclave execution," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 108–114.

[52] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," in *Proceedings of the 23st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Williamsburg, VA, Mar. 2018.

[53] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2019.

[54] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[55] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. V. Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-resistant CPUs," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[56] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin, "SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2017.

[57] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious Multi-Party Machine Learning on Trusted Processors," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[58] O. Ohrimenko, C. F. Manuel Costa, S. Nowozin, A. Mehta, F. Schuster, and K. Vaswani, "SGX-Enabled Oblivious Machine Learning," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[59] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace: Oblivious Memory Primitives from Intel SGX," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[60] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A Data Oblivious File System for Intel SGX," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[61] P. Zhang, C. Song, H. Yin, D. Zou, E. Shi, and H. Jin, "Klotski: Efficient Obfuscated Execution against Controlled-Channel Attacks," in *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, Apr. 2020.

[62] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in Darkness: Return-oriented Programming against Secure Enclaves," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[63] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The Guard's Dilemma: Efficient Code-Reuse Attacks against Intel SGX," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[64] Intel, "3rd Gen Intel Xeon Scalable processors," https://www.connection.com/~/media/pdfs/brands/i/intel/intel-icelake-ds.pdf?la=en.

[65] ——, "Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation (Revision: 4.1)," 2018.

[66] Greg, "SGX Attestation results in CONFIGURATION_NEEDED," 2018, https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/798777.

[67] ——, "GROUP_OUT_OF_DATE - what is the most recent microcode version?" 2018, https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/755769.

[68] Clemens Hammacher, "Liftoff: a new baseline compiler for webassembly in v8," 2018, https://v8.dev/blog/liftoff.

[69] A. Shilov, "Intel's New Core and Xeon W-3175X Processors: Spectre and Meltdown Security Update," 2018, https://www.anandtech.com/show/13450/intels-new-core-and-xeon-w-processors-fixes-for-spectre-meltdown.

[70] P. Kocher, "Spectre Mitigations in Microsoft's C/C++ Compiler," 2018, https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html.

[71] "Lighttpd," 2003, https://www.lighttpd.net/.

[72] "libjpeg," 1991, https://libjpeg.sourceforge.net/.

[73] "SQLite," 2000, https://www.sqlite.org/index.html.

[74] "PolyBench," 2015, http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[75] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai, "Smashex: Smashing sgx enclaves using exceptions," in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Republic of Korea, Nov. 2021.

[76] Intel, "The latest security information on Intel products," 2021, https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00548.html.

[77] Open Enclave, "Open Enclave SDK Elevation of Privilege Vulnerability," 2021, https://github.com/openenclave/openenclave/security/advisories/GHSA-mj87-466f-jq42.

[78] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[79] E. Bauman, H. Wang, M. Zhang, and Z. Lin, "SGXElide: Enabling Enclave Code Secrecy via Self-modification," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*, Vienna, Austria, Feb. 2018.

[80] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[81] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[82] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

[83] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch, "SGX-LKL: Securing the Host OS Interface for Trusted Execution," *CoRR*, vol. abs/1908.11143, 2020. [Online]. Available: http://arxiv.org/abs/1908.11143

[84] S. Arnautox, B. Tarch, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux containers with Intel SGX," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[85] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux applications with SGX enclaves," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

## A Additional Related Work

**In-enclave loader.** Researchers study in-enclave loaders to enhance the security and deployability of Intel SGX. For the security, several loaders leverage randomization and encryption. SGX-Shield [43] loads SGX applications while

```
# Varys
BB:
  ...
  jmp loop.header
loop.body:
  call varys_check
  ...
  incq %rcx
loop.header:
  call varys_check
  ...
  cmpq $100, %rcx
  jbe  loop.body
loop.end:
  call varys_check
  ...
```

```
# T-SGX
BB:
  ...
  leaq loop.header(%rip), %r15
  jmp  springboard.next
loop.body:
  ...
  incq %rcx
loop.header:
  ...
  cmpq $100, %rcx
  jbe  loop.body
  leaq loop.end(%rip), %r15
  jmp  springboard.next
loop.end:
  ...
```

**Figure 9:** The comparison of Varys and T-SGX on a loop structure.

enforcing fine-grained ASLR. VC3 [78] and SGXElide [79] deploy encrypted SGX code while decrypting it within an enclave. Obfuscuro [31] obfuscates SGX code with Oblivious RAM. For the deployability, some loaders abstract the interface between SGX code and the outside. Ryoan [80] implements a two-way sandbox to securely execute untrusted code inside an enclave. Haven [81], Graphene-SGX [82], and SGX-LKL [83] run a library OS inside an enclave to executed unmodified programs. Similarly, SCONE [84] abstracts system call interfaces and Panoply [85] abstracts POSIX interfaces to run unmodified programs with SGX. Unlike these approaches, PRIDWEN focuses on how to instrument SGX applications according to the hardware features to improve their security.

**SGX and Wasm.** To the best of our knowledge, there are a few initial efforts to execute Wasm interpreters inside an enclave. Rust-SGX [36] can be configured to use Wasm as a backend. Se-Lambda [37] executes serverless functions written in Wasm inside an enclave. Also, Intel and Red Hat are developing Wasm runtime for SGX [38, 39]. However, unlike PRIDWEN, these approaches only run existing Wasm interpreters without improving their functionalities.

**Wasm instrumentation.** Other studies also instrument Wasm binaries to detect security attacks. SEISMIC [40] instruments Wasm binaries to inject an inline monitor for detecting cryptojacking. Wasabi [41] is a Dynamic Binary Instrumentation (DBI) tool that statically instruments Wasm binaries to inject hooks and dynamically runs JavaScript-based analysis code on them to find potential bugs. However, unlike PRIDWEN, they do not consider instrumenting native binaries compiled from Wasm binaries, which is necessary to adopt low-level security mitigations sensitive to machine code.

## B Loop Comparison: Varys vs. T-SGX

The comparison of Varys and T-SGX on a loop structure is shown in Figure 9.