Talk: Debugging the Un-Debuggable: Advanced Debugging Techniques for Microarchitectural Security Tooling

Anna Pätschke
University of Luebeck

Daan Vanoverloop
DistriNet, KU Leuven

Jan Wichelmann
University of Luebeck

Jo Van Bulck
DistriNet, KU Leuven

Abstract

System security research often involves the development of custom code for building program analysis tools or implementation of side-channel leakage mitigations. In various cases, those specialized contexts also render the usage of conventional debugging tools, such as GDB or integrated IDE debuggers, unusable. Examples for this are the development of code that runs on specialized hardware platforms, such as the Proteus RISC-V core, where debugging must be conducted by analyzing low-level hardware signal traces rather than using interactive support, or when custom binary instrumentation or hardware protection like Intel SGX interferes with the debugger's operation. Therefore, the aforementioned cases demand new debugging strategies.

In this talk, we present various ways to trace programs at runtime and collect data in a unified format. We furthermore discuss how this data can be used for offline analysis to restore debugging functionality that was lost, or use it to check the security and conformance of programs. While we focus on the examples used throughout our projects in this talk, they represent only a fraction of possible solutions. This talk is intended to also spark further discussion about other tools and techniques, from small self-written scripts to best practices for leveraging larger, mature analysis frameworks. All proposed tooling is collected in a publicly available GitHub repository[1].

1 Beyond Breakpoints: A Deep Dive into Unconventional Debugging Techniques

The field of system security involves a wide range of research areas including the development of side-channel analysis tools, building hardening or verification frameworks, or finding attack vectors. All those areas also include the need for bug finding during the development of tooling. Another common theme among those tasks is the need for *deep program manipulation*. These research tasks often include writing software that operates far outside the bounds of typical software development, e.g., through binary instrumentation, modification of kernels, or prototyping hardware security extensions on custom RISC-V cores like Proteus [1]. Those tasks frequently preclude the use of traditional debugging tools.

For example, during development of compile-time side-channel mitigations, broken programs may be generated due to buggy transformation during or after compilation. To debug such issues, being able to step through the program, jump to breakpoints and observe register values is helpful to pinpoint the issue. However, this is not possible when using academic RISC-V cores that don't include a debug interface, or when the instrumentation interferes with the debugger.

A different use case is when debugging is disabled by design, for example, on Intel SGX production enclaves. Using side-channel leakage and temporal control through interrupts and page faults to restore some debugging functionality can help foster attack research, and analysis of side-channel leakage can help validate the efficacy of mitigations [5].

When conventional debugging fails, researchers need to find alternative strategies, which often includes writing new tooling for debugging purposes. To limit the time spent on that and foster shared knowledge, we introduce a collection of debugging techniques we have repeatedly used and refined across research projects.

The debugging process involves two distinct phases: generating data and analyzing data (see Fig. 1). Data generation and tracing can range from extracting low-level hardware signals, to software-level execution traces generated with dynamic binary instrumentation frameworks like Intel Pin [3]. We offer a systematic approach for unifying different information (trace) sources within an established shared format. For that, we choose to convert the collected data to Value Change Dump (VCD) files, a file format commonly used in hardware design to represent low-level hardware signals. This allows us to reuse existing open-source analysis tools, such as GTKWave and wellen. The data analysis tasks include a) *conformance*, does the program (or tool) behave as intended

[1] https://github.com/syssec-debugging-tools/catalog

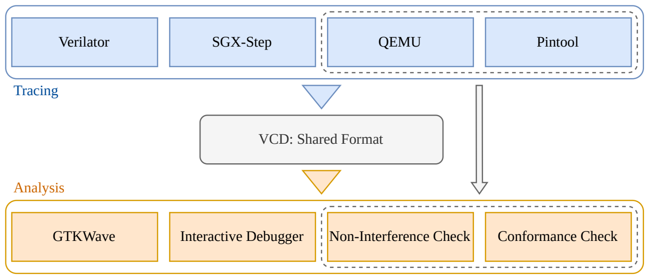

Figure 1: Overview of different stages of the adjusted debugging process: Tracing output gets converted into a shared format and is then passed on into different analysis modules. Shortcuts from specific trace formats to analysis modules are enabled where possible (as shown by the dotted gray boxes).

or does the transformed program produce the same semantic results as the original; and b) *non-interference* [2], are the observations leaked by the transformed program independent of its secret inputs?

1.1 Generating Data: Different Tracers

Pinpoint: pintool-based tracer. A versatile pintool that can trace x86-programs even if they contain `int3` instructions or other features that are incompatible with debuggers. Its features can be activated on demand and include tracing memory writes, register states, system calls, function arguments, and instruction counts and opcodes. To manage the data volume, it can be configured to trace only interesting functions or offsets, or skip a specified number of instructions.

Fast QEMU-based tracer. For full-system analysis, we leverage QEMU[2]. This tool hooks into the Tiny Code Generator (TCG) internals to enable lightweight, high-performance tracing of guest executions.

Strace debugging. A simple but effective hack for exfiltrating data when other methods fail. By instrumenting the target to perform a system call with many unused parameters, we can pack arbitrary register values into the syscall's arguments and read them from the tracer. Similarly, `sgx-tracer`[3] uses `ptrace` to intercept enclave loading and dump memory.

Verilator tracer. Custom processor cores written in Hardware Description Languages (HDLs) like SystemVerilog can be compiled using Verilator[4], a tool to generate cycle-accurate simulators. These simulators can be configured to emit VCD files that describe the changes of both architectural and microarchitectural state of the core throughout the execution. In this talk we focus on Proteus [1], an extensible RISC-V core for prototyping hardware extensions.

[2]https://www.qemu.org/
[3]https://github.com/pandora-tee/sgx-tracer
[4]https://www.veripool.org/verilator/

SGX-Step-based tracer. To help with attack research on Intel SGX enclaves, a tracing tool based on the SGX-Step framework [4] extracts page-granular memory accesses from an enclave at maximal temporal resolution. The resulting data can be analyzed with interactive tools, or by programmatically correlating leakage patterns to input data [5].

1.2 Analyzing Data: Processing Tools

GTKWave. Without any additional processing, VCD traces can already be analyzed in detail using a waveform viewer like GTKWave. However, pinpointing specific leakages or bugs is hindered by the level of detail provided.

Interactive debugger. A common problem with static VCD traces is the loss of interactivity. We present how low-level hardware traces can be converted from a pure feature-rich GTKWave notion to an accessible debugger-like walk-through of the program. Similar to record-and-replay approaches taken by other tools, this restores the feel of a standard debugger, but on offline data. This is especially useful when matching low-level traces with the corresponding instructions in the disassembled binary of the source program.

Conformance checks. Checking the semantic conformance of a new implementation or an instrumented version of an implementation is done by measuring differences to the expected outcome. On a low level, this includes the comparison of memory accesses, register values, and other (micro)architectural states to reason about invariants that should hold.

Non-interference checks. When implementing side-channel mitigations, we not only need to verify the conformance of the mitigation, but also its efficacy. We generate traces for the same program with different secret inputs, and check for differences in the attacker-observable signals to ensure the absence of side-channel leakage.

References

[1] M. Bognar et al. Proteus: An Extensible RISC-V Core for Hardware Extensions. In *RISC-V Summit Europe*, 2023.

[2] D. Denning and P. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 1977.

[3] C. Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[4] J. Van Bulck et al. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX*, 2017.

[5] D. Vanoverloop et al. TLBlur: Compiler-Assisted Automated Hardening against Controlled Channels on Off-the-Shelf Intel SGX Platforms. In *USENIX Security*, 2025.

Acknowledgements. This work was partially supported by the Research Fund KU Leuven, and the Cybersecurity Research Program Flanders, and by Deutsche Forschungsgemeinschaft (DFG) through the ReTEE project.