Automated Reasoning About Memory Accesses on Systems-on-Chip

Ben Fiedler
ETH Zürich
Zürich, Switzerland

Samuel Gruetter
ETH Zürich
Zürich, Switzerland

Timothy Roscoe
ETH Zürich
Zürich, Switzerland

Modern hardware platforms have grown enormously complex on multiple levels: the CPU, including its memory translation and protection units, has gained a tremendous amount of features over the last 30 years. Different operating modes and mechanisms for vertical (x86 rings, Arm execution levels, RISC-V modes, etc.), and horizontal (Intel's TDX, Arm's TrustZone, Arm's CCA, RISC-V's PMP, etc.) isolation exist, in addition to other hardware mechanisms that focus solely on memory protection (page tables, nested paging, memory keys, etc.). These mechanisms vary in subtle ways in their implementation between different processor models, even of the same vendor, which results in hard-to-maintain, and difficult-to-verify systems code to correctly utilize them.

Despite the obvious importance of these protection mechanisms, and therefore the need for correctly programming them, system software components that manage these different contexts are not designed coherently, but piecemeal: the hypervisor, realm monitors, system management firmware, and OS kernels form a *de-facto OS* [1]. The result is a slew of vulnerabilities which compromise a system by exploiting the incoherent design between different components [2, 3, 5]. Even the combination of two simple mechanisms already leads to significant coordination challenges between different parts of system software [4].

The central problem of modern hardware is the sheer complexity of the combination of these mechanisms, and this complexity is still increasing. At the abstract level, the individual components of a platform are restricted in their complexity: they receive a request to handle from an upstream component, potentially modify it and either respond to the upstream or route the request further downstream. When combining these components, each tasked with implementing a specific part of the isolation mechanisms mentioned above, the complexity skyrockets and vulnerabilities emerge.

In order to automatically detect, derive, and hopefully also prevent these vulnerabilities, we formally specify the relevant parts of modern platforms, and automatically analyze the resulting model for potential vulnerabilities using SMT solvers.

To validate our approach, we started by manually encoding existing vulnerabilities as SMT queries. From here on, we focused on designing a formal model, in the form of a domain-specific language, for specifying the capabilities and combination of individual hardware components.

We aim to discover new vulnerabilities in three steps:

1. Model the memory addressing behavior of each individual component in a machine according to its documentation, together with the associated interconnect.
2. Encode security properties we want the system to uphold. These could be classical properties, such as confidentiality, integrity, or non-interference, but they could also be informed by existing vulnerabilities. Additionally, we can also encode assumptions about the behavior of software running on other components here (e.g. "this firmware will only use the top third of DRAM").
3. Combine our hardware model, assumptions, and security property into a query that asks the SMT solver to find a scenario where the security property does not hold. The SMT solver's output can then be used to construct a possible attack.

We don't expect this procedure to be strictly linear when analyzing a given hardware platform. While the initial model of the hardware constructed in step 1 is independent of the targeted security properties, we expect there to quite some back-and-forth between steps 2 and 3, refining assumptions and validating potential attacks.

Once an attack is found, we extract a complete trace of operations which affect the system, which also allows us to investigate how to prevent it. Examples include correcting the configurations of some hardware components or modifying them, or targeted verification of systems software.

We present a prototype of the proposed analysis toolchain, which is able to encode and analyze simplified SoCs and exploits. Our current focus is two-fold: specifying real-world SoCs with existing vulnerabilities in order to further validate our approach, as well as specifying a number of existing platforms (e.g. Raspberry Pi, NXP i.MX6/8, STM32) based on their technical documentation to find new vulnerabilities, or prove their absence.

Future work includes figuring out whether we can classify properties and/or attacks that we discover to define the set of vulnerabilities which our approach find. Furthermore, we could explore whether there exist universal hardware design principles that would prevent these vulnerabilities *a priori*.

Eurosys'25, Mar 30–3 April 2025, Rotterdam, Netherlands
2025.

References

[1] Ben Fiedler, Daniel Schwyn, Constantin Gierczak-Galle, David Cock, and Timothy Roscoe. Putting out the hardware dumpster fire. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 46–52, Providence RI USA, June 2023. ACM.

[2] Trammell Hudson and Larry Rudolph. Thunderstrike: EFI firmware bootkits for Apple MacBooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–10, Haifa Israel, May 2015. ACM.

[3] Enrique Nissim and Krzysztof Okupski. AMD Sinkclose: Universal Ring -2 Privilege Escalation, August 2024.

[4] NIST. CVE-2021-36133.

[5] NIST. CVE-2021-44149.