Principled Symbolic Validation of Enclaves on Low-End Microcontrollers

Gert-Jan Goossens
DistriNet, KU Leuven
Leuven, Belgium
gertjan.goossens.1603@gmail.com

Jo Van Bulck
DistriNet, KU Leuven
Leuven, Belgium
jo.vanbulck@kuleuven.be

Abstract—**Recent advancements in trusted execution environments (TEEs) provide strong isolation guarantees for hardware-protected enclaves within a shared address space. The advent of commercial solutions like Intel SGX on high-end processors has spurred a growing open-source ecosystem of enclave shielding runtimes, along with research into symbolic execution tools for detecting elusive interface sanitization bugs. However, despite their inherent similarities and shared vulnerabilities, automated validation of enclaves on low-end embedded platforms remains largely unexplored.**

This paper ports Pandora, a symbolic execution tool originally designed for principled validation of high-end Intel SGX enclaves, to the Sancus research TEE for 16-bit MSP430 microcontrollers. We introduce a TEE hardware abstraction layer and extend Pandora's symbolic memory model to support non-contiguous Sancus enclaves. Our evaluation across different runtimes and applications within the Sancus ecosystem demonstrates that Pandora autonomously re-discovers vulnerabilities that were manually patched over the last decade. Our work lays the foundation for automated validation of heterogenous enclaves and outlines directions for future work on interruptibility, real-time guarantees, and extensions to alternative MSP430 TEEs.

1. Introduction

Trusted execution environments (TEEs) protect data in use by isolating critical applications within hardware-enforced protection domains. Early TEE designs, such as Intel SGX, focused on fine-grained isolation of small *enclaves* within the virtual address space of an untrusted host process. Due to SGX's popularity in high-end processors, enclave-specific isolation challenges, such as pointer safety and register sanitization, have been widely studied [1]–[9]. However, recent industry trends on "confidential computing" [10] are shifting toward coarser-grained *lift-and-shift* isolation, protecting entire virtual machines (VMs) in untrusted cloud environments.

At the same time, the rise of the Internet of Things (IoT) has made small, resource-constrained microcontrollers without advanced hardware support for VM-based isolation ubiquitous in households and industry. These embedded IoT platforms often lack virtual memory, privilege rings, or hardware virtualization, necessitating specialized memory isolation mechanisms. Several low-end TEE research prototypes [11]–[16], as well as some commercial microcontrollers [17]–[19], include lightweight hardware support to isolate small enclave regions in the shared physical address space. Security analyses of these low-end enclaves [5], [17], [20], [21] have shown that they are prone to the same types of interface sanitization oversights as their high-end Intel SGX counterparts. However, despite this striking similarity and the existence of a sizable open-source Intel SGX software ecosystem and validation tools [1]–[4], [7], the automated validation of enclaves on low-end embedded platforms remains largely unexplored.

To address this gap, this paper adapts Pandora [1], an open-source symbolic execution tool based on `angr` [22] and originally designed for the principled validation of high-end Intel SGX enclaves, by introducing a hardware abstraction layer for flexible extensibility across different enclave architectures. Using the mature Sancus [11] research TEE for low-end 16-bit MSP430 microcontrollers as a case study, we extensively refactor Pandora's codebase to eliminate x86-specific SGX dependencies and generalize its symbolic memory model and taint-tracking mechanisms to support non-contiguous enclave memory layouts and architecture-specific enclave loaders. Adhering to Pandora's *truthful* symbolic execution principles, we seamlessly integrate `angr`'s MSP430 back-end, provide extensible hooks for Sancus-specific cryptographic and enclave instructions, and precisely model Sancus's hardware-enforced access control semantics and exception behavior. Furthermore, we extend Pandora's *pluggable* vulnerability detection to the MSP430 Sancus TEE, enabling automated detection of elusive pointer-sanitization flaws and control-flow hijacking vulnerabilities in Sancus enclaves through human-readable HTML reports.

We evaluate our Pandora port through a comprehensive unit-test framework, including 30 crafted assembly test cases for control-flow and pointer sanitization vulnerabilities, along with 13 additional Sancus unit-test enclaves written in C. Furthermore, we demonstrate that Pandora autonomously reproduces over 6 subtle vulnerabilities previously identified through painstaking manual analysis [5], including critical issues in various versions of Sancus's compiler runtime [11], [23], applications [24], and derived architectures [25].

More broadly, our work establishes a foundation for the automated validation of enclave software across heterogeneous architectures and outlines future directions on interruptibility, real-time guarantees, and extensions to alternative MSP430 TEEs [12], [13], [17], [18], [26].

Contributions. In summary, our main contributions are:

- We design a hardware abstraction layer to support architecture-agnostic, truthful symbolic execution of enclaves across heterogeneous TEEs.

- We accurately implement hardware semantics and symbolic enclave loading for the Sancus research TEE on low-end MSP430 microcontrollers.
- We evaluate our port through extensive unit tests and autonomous reproduction of vulnerabilities across Sancus applications and compiler runtimes.

Open Science. To ensure reproducibility and encourage future research, all our modifications to Pandora, along with our evaluation enclaves, have been merged into the upstream Pandora open-source repository available at https://github.com/pandora-tee/.

2. Background and Related Work

TEEs. Hardware-based trusted execution environments enhance processors with primitives for isolation and attestation of critical code and data [10]. Recent years have seen the emergence of various architectures that implement TEE protection at different isolation granularities. Cloud-based TEEs, such as AMD SEV and Intel TDX, provide coarse-grained isolation for entire virtual machines, while enclave-based TEEs, like Intel SGX, offer fine-grained protection within the same address space. In the latter design, hardware-isolated *enclave* regions are embedded within an untrusted host's address space. Although attackers can address enclave memory, the CPU ensures it remains inaccessible to all code outside the protected region. When the CPU executes within the enclave, on the other hand, access is granted to the entire address space, enabling efficient access to input and output buffers but potentially exposing enclave software to confused-deputy pointer vulnerabilities [5].

Sancus. The mature Sancus [11], [23] TEE research prototype extends the openMSP430 softcore, a small microprocessor with a flat 16-bit single address space, designed for embedded devices like pacemakers and sensor nodes. Sancus enclaves consist of a contiguous, read-only code section and a single data section for secrets. A lightweight hardware memory access-control mechanism ensures that an enclave's data section is only accessible when executing within the corresponding text section. To simplify secure enclave development, Sancus provides a modified LLVM C compiler that automates low-level tasks such as maintaining a secure call stack and handling enclave entry and exit. Carefully crafted entry and exit stubs are automatically inserted into binaries, making them critical to Sancus's security: any flaw in these stubs affects all enclaves built with the toolchain [5].

Since its original release over a decade ago, the Sancus TEE research prototype has been maintained as an active open-source project, driving numerous embedded applications and extensions [11], [20], [21], [24], [25], [27]–[30] and inspiring similar low-end single-address-space enclave architectures [12], [14]–[16], [26].

Pandora. Pandora [1] is an open-source extensible validation framework built on top of the popular `angr` [22] library. In contrast to other SGX validation tools [2]–[4],

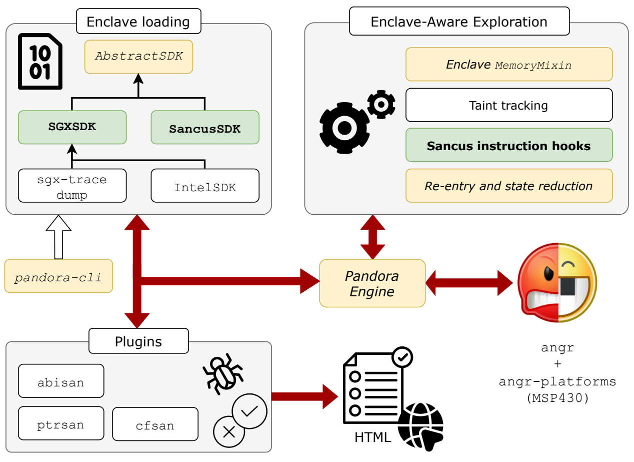


Figure 1. Overview of our main changes to the Pandora software architecture. Newly added components are highlighted in green (bold) and modified components are in yellow (italics).

[7], Pandora explicitly aims for *truthful* symbolic execution, where the hardware is mimicked as close as possible, allowing to meticulously validate low-level enclave runtime entry/exit behavior and initialization logic. Pandora extends `angr`'s functionalities with accurate semantics for SGX-specific instructions, an enclave-aware symbolic memory model, and powerful techniques such as taint tracking of attacker inputs.

Pandora features an extensible, plugin-based design for vulnerability detection, where individual plugins implement detection logic by enforcing high-level invariants. These invariants are checked during the symbolic exploration by subscribing to specific "breakpoints" exposed by Pandora's enclave-aware memory model. For instance, Pandora's `ptrsan` plugin transparently validates that all attacker-tainted memory addresses always resolve outside the enclave, thereby ensuring the absence of evasive confused-deputy attacks [5]. Similarly, `cfsan` excludes control-flow hijacking attacks by restricting attacker-tainted jump targets to fall outside the enclave, while `abisan` ensures that CPU registers are properly initialized and cleared upon enclave entry and exit. After symbolic exploration, Pandora generates detailed human-readable HTML reports for each plugin, including contextual information such as register states and disassembly.

While Pandora can load and symbolically execute *any* SGX enclave, regardless of its shielding runtime, it remains tightly coupled to Intel SGX. Its codebase depends on low-level SGX-specific details such as binary format loading, memory layout structures (e.g., thread-control structure), and SGX-specific x86 instructions. Hence, symbolic validation of enclaves for other TEEs, like Sancus, remains an open challenge.

3. Implementation

Our goal was to design an extensible TEE hardware abstraction layer that exposes Pandora's core symbolic-execution infrastructure to support principled validation of single-address-space enclaves across heterogeneous architectures, while preserving truthful semantics of low-level TEE-specific hardware behavior.

We provide an overview of the main changes to Pandora's software architecture in Figure 1. Our modifications primarily focus on the binary loading subsystem and the enclave-aware symbolic exploration components. Notably, we did not need to modify the `ptrsan` and `cfsan` plugins, which express high-level, architecture-independent security invariants. We leave extension of the architecture-dependent `abisan` plugin as future work

Enclave Loading. In Intel SGX, enclaves are included as dynamically linked ELF libraries, whereas in Sancus, one or more enclaves are directly embedded within the entire firmware binary. Moreover, SGX employs a complex multi-stage loading process [31] to create the enclave's virtual address space, including hardware-managed data structures. We found references to SGX-specific data structures, such as the thread-control structure [31], scattered throughout Pandora's codebase. Therefore, we refactored binary loading and encapsulated SGX-specific behavior in an abstract base class. This modular design simplifies extending Pandora with custom enclave loaders.

We implemented a `SancusSDK` enclave loader that automatically detects MSP430 binaries and extracts enclave boundaries by detecting the custom ELF sections for enclave text and data regions, added by the Sancus compiler. `SancusSDK` also takes care to initialize the symbolic CPU register state, setting the program counter to the single enclave entry point at the start of the text section and marking all other MSP430 registers as attacker-tainted. A further important consideration is that in Intel SGX an enclave has to be a contiguous area in virtual memory, whereas a Sancus enclave can be split into two separate regions in memory. Since Pandora's original symbolic memory model was limited to a single contiguous enclave address range, we extended it to support multiple contiguous regions, ensuring breakpoints trigger whenever a symbolic pointer overlaps with *any* of these regions.

Instruction Hooks. We rely on the open-source *angr-platforms* repository[1] to lift standard MSP430 instructions to `angr`'s internal VEX intermediate representation. However, since `angr`'s built-in Capstone disassembler lacks MSP430 support, `SancusSDK` mitigates this by externally invoking `msp430-objdump` to disassemble the binary and parsing the output for inclusion in reports and debugging. We furthermore transparently detect and hook Sancus-specific instructions that are not part of the standard MSP430 instruction set upon enclave loading.

Instruction hooks must accurately reflect the effects of Sancus-specific instructions on the symbolic execution state. Some instructions, like `enable`, are typically not called within enclaves and can be safely treated as no-ops. We implement `disable` by modifying the state to halt execution of the current symbolic path. The `verify` instruction, used for local attestation between multiple enclaves, is skipped, as our Pandora port does not yet support multi-enclave linking (Section 5). This also allows to statically return zero for `get_id` and `get_caller_id`, indicating the enclave was invoked by untrusted code. The `wrap` and `unwrap` instructions are partially implemented by setting authenticated encryption/decryption results as unconstrained symbolic values. Future work could explore executing these cryptographic operations concretely using a fixed or user-provided master key. Lastly, since our Pandora port does not simulate interrupts, we treat `clix` (interrupt-disable for `x` cycles [30]) as a no-op. However, we accurately simulate exceptions for illegal nested `clix` invocations, as they are used to trigger hardware exceptions in Sancus's runtime `ASSERT` macros.

1. https://github.com/angr/angr-platforms

Enclave Exit. Unlike Intel SGX, where enclaves are entered and exited via dedicated `EENTER/EEXIT` instructions, Sancus uses regular control flow instructions like `call` or `jmp` for *implicit* transitions. Since Pandora originally depended on explicit `EEXIT` hooks to manage symbolic execution paths, we refactored the engine to query architecture-specific classes, such as `SancusSDK`, to identify enclave exit points. Upon detecting an exit, Pandora's symbolic explorer now uses `angr`'s stashing mechanism to halt or re-enter execution paths as needed.

Hardware Exceptions. Pandora originally supported only x86 page-level access control. We extended it to accurately model Sancus's fine-grained enclave memory protection rules. Specifically, `SancusSDK` registers the enclave data section as non-executable and sets extra breakpoints to enforce read- and execute-only permissions on the text section. This is crucial when executing Sancus compiler entry/exit stubs (cf. Section 4.2), which may intentionally write to the text section to trigger runtime exceptions and abort execution upon detecting interface violations.

4. Evaluation

We evaluated our Pandora port using both handwritten unit tests and existing applications and compiler runtimes within the Sancus ecosystem. Although we did not uncover new vulnerabilities, our evaluation clearly shows that Pandora can autonomously reproduce known vulnerabilities in existing Sancus applications and compiler runtimes that have been manually discovered and patched over the past decade.

4.1. Unit Test Framework

To validate expected behavior and refine our Sancus-specific symbolic execution engine, we developed an extensive unit-test framework. This includes 30 hand-crafted assembly test cases for control-flow and pointer sanitization vulnerabilities, as well as 13 additional Sancus unit-test enclaves written in C. The latter assess Pandora's handling of complete application enclaves, including complex compiler-generated runtime stubs, while the handcrafted assembly tests allow for precise modeling of specific vulnerabilities without compiler-induced overhead.

Appendix A details the Sancus unit tests for Pandora's `cfsan` and `ptrsan` vulnerability-detection plugins.

4.2. Compiler Runtime Entry and Exit Stubs

Validating compiler-generated enclaves is an essential goal. The critical entry and exit stubs inserted into every enclave consist of carefully crafted, hand-written assembly

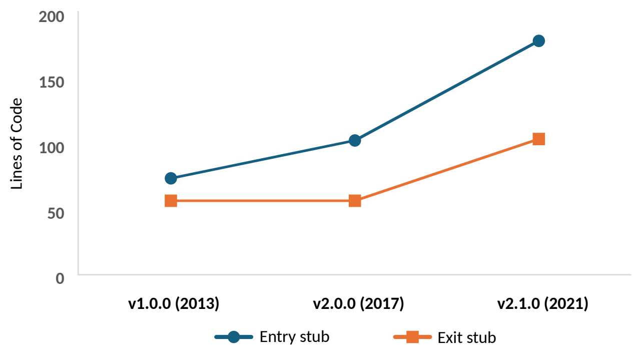


Figure 2. Growth of Sancus enclave runtime stubs in lines of code.

code that is a prime target for attackers and prone to subtle sanitization oversights. Figure 2 shows the increasing size and complexity of these stubs across Sancus versions, reflecting trends seen in Intel SGX shielding runtimes [9]. Previous research [5] identified numerous control-flow hijacking and pointer validation vulnerabilities in earlier Sancus stub versions. We ran Pandora on a test C enclave across all stub versions and confirmed that it autonomously rediscovered all issues (cf. Table 1).

Reproduced Issues. The cfsan plugin flagged a critical issue as a *symbolic unconstrained tainted jump target*, indicating control-flow hijacking to any address within or outside the enclave across the entire 16-bit address space, confirming previous findings [5]. Indeed, the HTML report (see Figure 5 in Appendix C) correctly shows that no constraints are placed on the r7 register, which stores the continuation address for enclave exit. Since the stub does not verify if this address belongs to the caller, an attacker can trick the enclave into jumping to any address, even within the enclave itself.

Both the cfsan warning and the ptrsan critical issue originated from the __sm_ret_entry stub. This stub was previously reported [5] to suffer from a vulnerability where an enclave could be forced to "return" from a callback that was never executed, causing the enclave to pop under the stack and load incorrect values into registers. The HTML reports (see Figures 4 and 7 in Appendix C) indicate that the final ret instruction would return to address 0x0, which is indeed the initialized value of enclave memory. The cfsan warning correctly flagged this as a *concrete return target in non-executable memory*, while ptrsan further detected a *non-tainted read outside enclave memory* at address 0x0.

Remaining Warnings. In the latest stub version v2.1.0, two ptrsan warnings remain due to an attacker-tainted pointer dereference in the enclave's logical entry-point jump table (__sm_table). This is expected behavior, as the index value in r6 is indeed attacker-controlled. However, Pandora downgraded this issue to a warning, as it autonomously determined that the attacker-provided index is properly constrained within the jump table bounds.

4.3. Sancus Applications and Libraries

The issues discovered in previous research [5] were not limited to Sancus's compiler stubs, but also span different applications and support libraries.

TABLE 1. Reported issues across Sancus stub versions.

	cfsan		ptrsan	
Version	**# warning**	**# critical**	**# warning**	**# critical**
1.0.0	1	1	2	1
2.0.0	1	1	2	1
2.1.0	0	0	2	0

4.3.1. Authentic Execution.

A first vulnerability was previously found in a Sancus extension [24], [27] designed for authenticated event-driven IoT applications. The enclave transparently decrypts and authenticates encrypted payloads while copying them inside. However, Pandora correctly detected that the payload input buffer pointer was not sanitized. Listing 5 in Appendix B shows the vulnerable code where sancus_unwrap_with_key is called with an *unsanitized* payload argument on Line 12. This function internally uses the aforementioned unwrap Sancus hardware instruction to decrypt the payload and writes it into the trusted input_buffer inside the enclave. Pandora successfully detected unconstrained reads and writes inside the enclave, autonomously identifying the vulnerability reported in prior work [5].

4.3.2. Soteria Loader Enclave.

Another vulnerability was discovered in a trusted loader enclave [25] developed for supporting lightweight code confidentiality and integrity. The vulnerable enclave is included in Listing 6 in Appendix B, with the vulnerability arising in Lines 3 through 6. The untrusted context passes an enclave layout struct to this enclave which is then immediately accessed inside sm_loader_load. This function does not execute any bounds checking on the values fetched from this struct. Pandora was able to discover these vulnerabilities indicating four unique critical *unconstrained read* issues. Manual inspection confirmed that these issues indeed stem from the unsanitized accesses to the attacker-controlled SancusModule argument exploited in prior work [5].

4.3.3. Insufficient Bounds Checks.

A final, particularly subtle vulnerability was found in the Sancus support library function, explicitly designed to ease interface sanitization of untrusted pointers. Listing 7 in Appendix B shows an enclave using this function. The vulnerable C macro sancus_is_outside_sm on Lines 1 to 6, aims to check if an *entire* buffer lies outside the enclave, but it only verifies the buffer's endpoints. This allows a large buffer that spans beyond the enclave's boundaries to incorrectly pass the check. Additionally, the function may also fail if the attacker-controlled buffer's length causes the address calculation to overflow and silently wrap around the address space. This highlights the complexities of correctly validating enclave software interfaces and the need for automated validation tools.

The ptrsan plugin, relying on Pandora's powerful symbolic enclave memory model and taint-tracking mechanism, fully autonomously reproduced this issue, which was previously reported through manual analysis in prior work [5]. Pandora also confirmed that this issue was correctly resolved in later Sancus support library versions.

5. Discussion and Future Work

Our Pandora port establishes a foundation for the automated validation of enclave software on low-end TEEs. In this section, we discuss limitations and opportunities for future work.

Validating Enclave Interactions. Both Intel SGX and Sancus support multiple enclaves. An important note here is that Intel SGX enclaves are completely isolated from each other. This is not necessarily the case for Sancus where enclaves can securely link with and call each other. However, our current Pandora port only validates single enclaves in isolation. When Pandora encounters binaries with multiple enclaves, it will default to the first enclave in the address space. For entirely isolated enclaves this is no problem but for enclaves that depend on other enclave code modules this may be a limitation. An interesting future research direction could extend Pandora to validate the secure linking and local attestation process of complex binaries with multiple interacting Sancus enclaves.

ABI Validation. A limitation of our current Pandora port is that the `abisan` plugin has not yet been adapted for the Sancus-MSP430 architecture. Unlike the main `ptrsan` and `cfsan` plugins, which enforce high-level, architecture-independent invariants, `abisan` operates at the lower level of CPU registers and is therefore intrinsically architecture-dependent. Consequently, `abisan` needs to be ported to ensure that MSP430 control and data registers are properly initialized and cleansed upon enclave entry and exit [17], [20].

Incomplete Instruction Hooks. A minor limitation is that not all hooks for Sancus instructions are entirely implemented. When encountering such instructions, Pandora may no longer adhere to the principle of *truthful* symbolic execution depending on the specific instruction. However, this is not a major limitation, as our existing base implementation of Sancus hooks can be easily extended to model additional behavior as needed.

Symbolic Execution Limitations. As this work is an extension of Pandora, it also inherits its limitations. This means that incomplete code coverage can be a side effect when encountering the path explosion phenomenon. This is however less of an issue for smaller embedded Sancus enclaves than for Intel SGX enclaves. Moreover, `angr` in itself is not guaranteed to be sound. Another limitation inherited from Pandora arises when encountering encrypted code. This can be overcome when provided with the decryption key but this is not the main goal of Pandora, as the developer can always run Pandora on the non-encrypted binary.

Heterogenous TEEs. Our adaptation of Pandora with an extensible hardware abstraction layer and our MSP430-based Sancus implementation opens the possibility for further TEE extensions. One promising direction could target the validation of hard real-time guarantees, for instance of the Aion [30] architecture which extends Sancus with enclave interruptibility and availability guarantees. However, possible future directions are not limited to Sancus only. As this work has done essential efforts in supporting truthful symbolic execution of embedded MSP430 programs, future developments could aim to support other enclave architectures developed on top of the MSP430 architecture, such as VRASED [12], [26] and SMART [13]. Especially for VRASED's remote attestation process, which is formally verified, would provide interesting results as research already indicated that gaps between formal models and its real-world implementation might exist, including insufficient interface validation in unverified glue code stubs [20].

6. Conclusion

Trusted execution environments have seen considerable developments in recent years. Among these developments, the ecosystem for many TEE architectures has been widely extended. Although TEEs provide strong, hardware-based isolation guarantees, developing secure enclave software remains challenging. While considerable research has been directed towards enclave software auditing and the development of automated validation techniques, virtually all of this work has focused exclusively on high-end TEE architectures like Intel SGX, neglecting low-end embedded TEEs. This work bridges that gap by extending Pandora, a mature open-source tool for validating SGX enclaves, to the Sancus research architecture for low-end enclaves on 16-bit microcontrollers.

Acknowledgments

We thank Fritz Alder for providing guidance in early stages of this research. This work was partially funded by the Research Fund KU Leuven, the Research Foundation – Flanders (FWO) via grant #1261222N, and by the Cybersecurity Research Program Flanders.

References

[1] F. Alder, L.-A. Daniel, D. Oswald, F. Piessens, and J. Van Bulck, "Pandora: Principled symbolic validation of Intel SGX enclave runtimes," in *45th IEEE Symposium on Security and Privacy (S&P)*, May 2024.

[2] T. Cloosters, M. Rodler, and L. Davi, "TeeRex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves," in *29th USENIX Security Symposium (USENIX Security 20)*, Aug. 2020, pp. 841–858.

[3] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "Coin attacks: On insecurity of enclave untrusted interfaces in SGX," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 971–985.

[4] P. Antonino, W. A. Woloszyn, and A. W. Roscoe, "Guardian: Symbolic validation of orderliness in SGX enclaves," in *Proceedings of the 2021 on Cloud Computing Security Workshop*, 2021, pp. 111–123.

[5] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *26th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2019, pp. 1741–1758.

[6] F. Alder, J. Van Bulck, D. Oswald, and F. Piessens, "Faulty point unit: ABI poisoning attacks on Intel SGX," in *36th Annual Computer Security Applications Conference (ACSAC)*, Dec. 2020, pp. 415–427.

[7] Y. Wang, Z. Zhang, N. He, Z. Zhong, S. Guo, Q. Bao, D. Li, Y. Guo, and X. Chen, "Symgx: Detecting cross-boundary pointer vulnerabilities of SGX applications via static symbolic execution," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2710–2724.

[8] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai, "Smashex: Smashing SGX enclaves using exceptions," in *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, 2021, pp. 779–793.

[9] J. Van Bulck, F. Alder, and F. Piessens, "A case for unified ABI shielding in Intel SGX runtimes," in *5th Workshop on System Software for Trusted Execution (SysTEX)*. ACM, Mar. 2022.

[10] C. C. Consortium, "Common terminology for confidential computing," https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/Common-Terminology-for-Confidential-Computing.pdf, 2022.

[11] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for IoT devices," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 3, pp. 1–33, 2017.

[12] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified Hardware/Software Co-Design for remote attestation," in *28th USENIX Security Symposium (USENIX Security 19)*, Aug. 2019, pp. 1429–1446.

[13] K. Eldefrawy, A. Francillon, D. Perito, and G. Tsudik, "Smart: Secure and minimal architecture for (establishing a dynamic) root of trust," in *NDSS 2012, 19th Annual Network and Distributed System Security Symposium*, 2012.

[14] R. De Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede, "Secure interrupts on low-end microcontrollers," in *25th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2014, pp. 147–152.

[15] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: a security architecture for tiny embedded devices," in *9th European Conference on Computer Systems (EuroSys)*. ACM, 2014, pp. 10:1–10:14.

[16] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *52nd ACM/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[17] M. Bognar, C. Magnus, F. Piessens, and J. Van Bulck, "Intellectual property exposure: Subverting and securing Intellectual Property Encapsulation in Texas Instruments microcontrollers," in *33rd USENIX Security Symposium*, Aug. 2024.

[18] Texas Instruments, "MSP code protection features," https://www.ti.com/lit/an/slaa685/slaa685.pdf, 2015.

[19] Microchip, "Codeguard security: Protecting intellectual property in collaborative system designs," http://ww1.microchip.com/downloads/en/DeviceDoc/70179a.pdf, 2006.

[20] M. Bognar, J. Van Bulck, and F. Piessens, "Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures," in *43rd IEEE Symposium on Security and Privacy (S&P)*, May 2022.

[21] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic," in *25th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2018, pp. 178–195.

[22] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[23] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *22nd USENIX Security Symposium*, 2013, pp. 479–494.

[24] J. Noorman, J. T. Mühlberg, and F. Piessens, "Authentic execution of distributed event-driven applications with a small TCB," in *13th International Workshop on Security and Trust Management (STM)*, 2017, pp. 55–71.

[25] J. Götzfried, T. Müller, R. de Clercq, P. Maene, F. Freiling, and I. Verbauwhede, "Soteria: Offline software protection within low-cost embedded devices," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 241–250.

[26] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the toctou problem in remote attestation," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2921–2936.

[27] G. Scopelliti, S. Pouyanrad, J. Noorman, F. Alder, C. Baumann, F. Piessens, and J. T. Mühlberg, "End-to-end security for distributed event-driven enclave applications on heterogeneous TEEs," *ACM Transactions on Privacy and Security*, vol. 26, no. 3, pp. 1–46, 2023.

[28] J. Van Bulck, J. T. Mühlberg, and F. Piessens, "VulCAN: Efficient component authentication and software isolation for automotive control networks," in *33rd Annual Computer Security Applications Conference (ACSAC)*, Dec. 2017, pp. 225–237.

[29] M. Bognar, H. Winderix, J. Van Bulck, and F. Piessens, "Microprofiler: Principled side-channel mitigation through microarchitectural profiling," in *8th IEEE European Symposium on Security and Privacy (EuroS&P)*, Jul. 2023.

[30] F. Alder, J. Van Bulck, F. Piessens, and J. T. Mühlberg, "Aion: Enabling open systems through strong availability guarantees for enclaves," in *28th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2021.

[31] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 86, 2016.

Appendix A. Unit Test Details

Our Pandora port includes an extensive unit test framework, including 15 and 15 crafted assembly test cases for control-flow and pointer sanitization vulnerabilities respectively, along with 13 additional Sancus unit-test enclaves written in C. We detail the tests for Pandora's `cfsan` and `ptrsan` vulnerability-detection plugins below, discussing elementary assembly enclaves that give a more intricate insight into the vulnerabilities discoverable with Pandora.

A.1. Control-Flow Sanitization Tests

One of the checks done by the `cfsan` plugin ensures that an enclave does not jump to an attacker tainted address that is not restricted to either the inside or the outside of the enclave. Listing 1 is an example of such a vulnerable enclave. The `br` instruction results in a jump to the provided address. The enclave will thus jump to `r15`. As there are no constraints imposed on the value of register `r15`, this will result in a jump to an attacker provided address. The `cfsan` plugin reports this as a critical issue: *Symbolic unconstrained tainted jmp target*.

Listing 2 illustrates an example of a jump constrained within the enclave. The example uses absolute addresses, although relative addresses (using labels) could be used as well. A subtlety in the `cfsan` plugin is that it always overapproximates the target instruction to be six bytes long, as this is the maximum length of an MSP430 instruction. This approach avoids the need to compute the size of the target instruction for each jump. The absolute addresses can be obtained with the help of the object dump tool. On Line 4 the address `0x6c1a`, an address inside the enclave, is compared against the register `r15`, using the `cmp` instruction. All paths that assume `r15` lower than this address will leave the enclave, with the `jl` instruction. The remaining paths will compare the register `r15` to value `0x6c1e` which is also an address inside the enclave. Similarly, all paths that assume `r15` greater than or equal to this address will leave the enclave (with instruction `jge`). The remaining paths will thus have `r15` constrained to values between `0x6c1a` and `0x6c1e`. Next the `br` instruction will jump to the address inside `r15`. This jump is thus constrained inside the enclave itself as all paths that assumed `r15` to be lower or greater than these addresses have left the enclave. This will be reported by the `cfsan` plugin as a warning: *Symbolic jmp tainted target in enclave memory*.

For the `cfsan` plugin in total 15 assembly test cases were written of which 8 cases test normal enclave behavior (so without reports generated by the `cfsan` plugin). The other cases test specific vulnerabilities detectable with the `cfsan` plugin. For all cases the `cfsan` plugin behaves as expected.

```
.text
__sm_foo_public_start:
enter_foo:
     br r15

__sm_foo_public_end:
     ret

.data
__sm_foo_secret_start:
__sm_foo_secret_end:
```

Listing 1. An enclave jumping to an attacker provided address.

```
.text
__sm_foo_public_start:
enter_foo:
     cmp #0x6c1a, r15
     jl end

     cmp #0x6c1e, r15
     jge end

     br r15

     nop ;Address 0x6c1a
     nop
     nop ;Address 0x6c1e
     nop
     jmp __sm_foo_public_end

__sm_foo_public_end:
end:
     ret

.data
__sm_foo_secret_start:
__sm_foo_secret_end:
```

Listing 2. An enclave jumps to an attacker tainted address constrained to the inside of the enclave.

A.2. Pointer Sanitization Tests

The `ptrsan` plugin checks invariants for every memory read and write. Listing 3 gives an example of a write to a non-attacker-tainted target address. On Line 4 the address `0x1000`, an address outside the enclave, will be put in `r15`. The `mov` instruction can move values between registers and memory locations. Then on Line 5 the value of `0x1` will be written to the address inside `r15`. As this address lies outside the enclave, a critical issue will be generated: *Non-tainted write outside enclave*.

Listing 4 presents an instance of an enclave that reads an attacker tainted address inside the enclave. This example acts in a similar way as Listing 2 where the attacker-tainted value inside `r15` will first be constrained to lie within the address range of the enclave. On Line 10, the value at the address stored in register `r15` will be moved to the register `r14`. This will be reported as a critical issue: *Unconstrained read*.

```
1  .text
2  __sm_foo_public_start:
3  enter_foo:
4       mov #0x1000, r15
5       mov #0x1, @r15
6       nop
7       nop
8       nop
9       jmp __sm_foo_public_end
10
11 __sm_foo_public_end:
12      ret
13
14 .data
15 __sm_foo_secret_start:
16      .space 64
17 __sm_foo_secret_end:
```

Listing 3. An enclave writing to a non-tainted address that may lie outside the enclave.

```
1  .text
2  __sm_foo_public_start:
3  enter_foo:
4       cmp #0x6c0c, r15
5       jl end
6
7       cmp #0x6c24, r15
8       jge end
9
10      mov @r15, r14
11      jmp end
12
13 __sm_foo_public_end:
14 end:
15      ret
16
17 .data
18 __sm_foo_secret_start:
19 __sm_foo_secret_end:
```

Listing 4. An enclave reading an attacker tainted address inside the enclave.

For the `ptrsan` plugin in total 15 assembly test cases have been developed of which 4 test the normal enclave behavior. The remaining 11 cases test specific vulnerabilities detectable with the `ptrsan` plugin. For all these test cases the `ptrsan` reported vulnerabilities as expected.

Appendix B.
Autonomously Reproduced Issues

In what follows, the vulnerable code snippets from earlier manual research "A Tale of Two Worlds" [5] are shown. Section 4.3 gives an in depth explanation of all the vulnerabilities throughout the entire Sancus ecosystem that were discovered in this research.

```
void SM_ENTRY(basic_enclave) __sm_handle_input(
    uint16_t conn_id,
            const void* payload, size_t len)
```

```
{
    if (conn_id >= SM_NUM_INPUTS)
        return;

    const size_t data_len = len - AD_SIZE -
    SANCUS_TAG_SIZE;
    const uint8_t* cipher = (uint8_t*)payload +
    AD_SIZE;
    const uint8_t* tag = cipher + data_len;

    uint8_t* input_buffer = alloca(data_len);
    if (sancus_unwrap_with_key(__sm_io_keys[
    conn_id], payload, AD_SIZE,
                               cipher, data_len,
     tag, input_buffer))
    {
        __sm_input_callbacks[conn_id](
    input_buffer, data_len);
    }
}
```

Listing 5. The vulnerable code that decrypts an unsanitized payload.

```
int SM_ENTRY(sm_loader) sm_loader_load(struct
    SancusModule *scm)
{
    size_t pstart  = (size_t)scm->public_start;
    size_t pend    = (size_t)scm->public_end;
    size_t pcstart = (size_t)scm->public_start;
    size_t pcend   = (size_t)scm->public_end;
    size_t i;
    int ret;

    // check boundaries
    if (pend < pstart  pcend < pcstart)
        return 0;

    // check sizes
    if ((pend - pstart) != (pcend - pcstart))
        return 0;

    //...
}
```

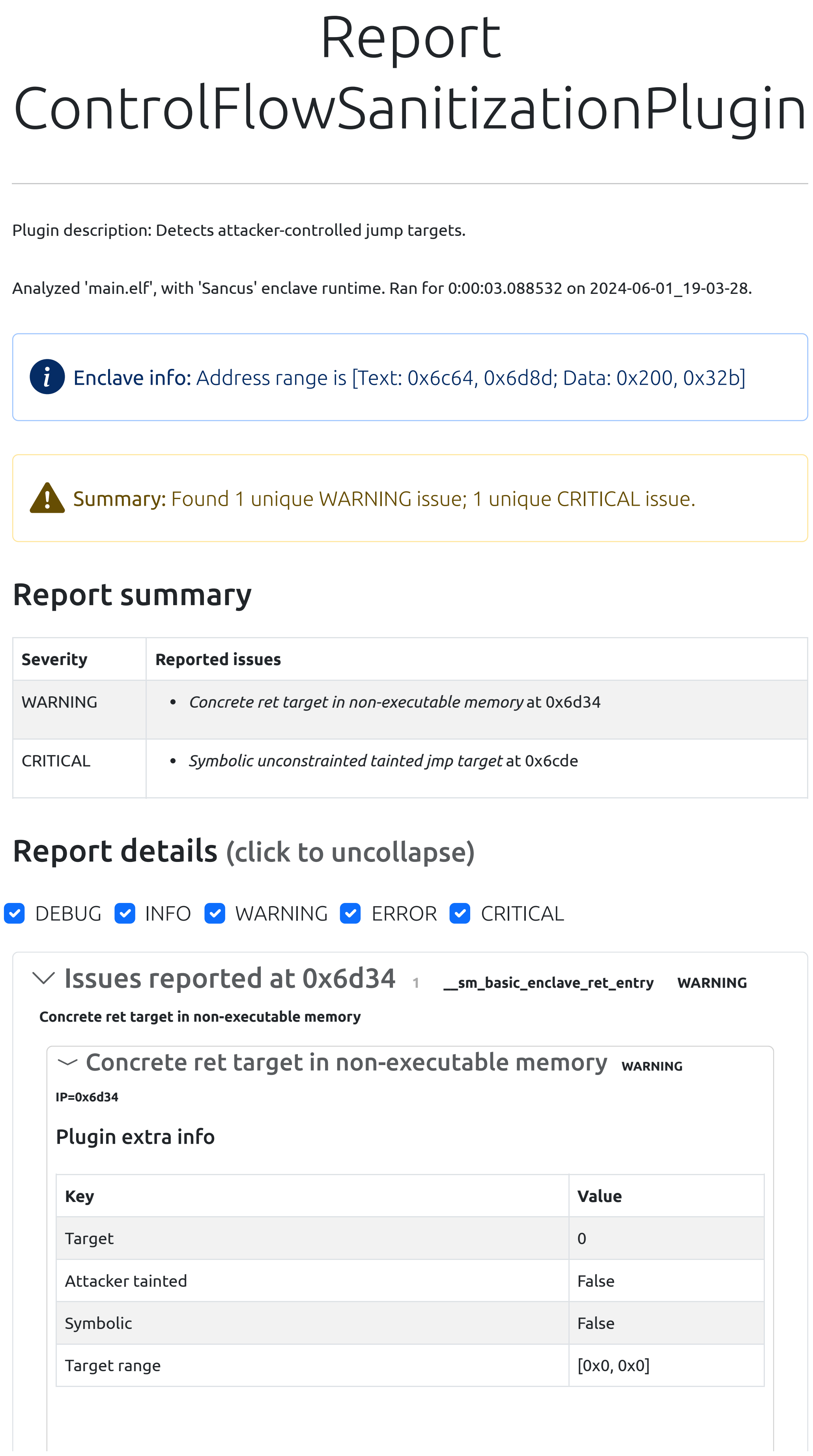
Listing 6. The vulnerable Soteria loader enclave.

```
#define __OUTSIDE_SM( p, sm ) \
    ( ((void*) p < (void*) &__PS(sm))   ((void*)
    p >= (void*) &__PE(sm)) ) && \
    ( ((void*) p < (void*) &__SS(sm))   ((void*)
    p >= (void*) &__SE(sm)) )

#define sancus_is_outside_sm_vulnerable(sm, p,
    len) \
    ( __OUTSIDE_SM(p, sm) && __OUTSIDE_SM((p+len
    -1), sm) )

void SM_ENTRY(basic_enclave)
    copy_data_from_buffer(int *buffer, int
    length)
{
    //vulnerable function
    if (!sancus_is_outside_sm_vulnerable(
    basic_enclave, buffer, length*2)) return;

    for (int i = 0; i < length; i++)
    {
        //access the data
        int result = buffer[i];
    }
    return;
}
```

Listing 7. An enclave checks arguments with the vulnerable `sancus_is_outside_sm`.

Appendix C. Generated Reports for Sancus Stubs v2.0.0

This appendix contains the complete `cfsan` and `ptrsan` reports generated for the Sancus entry and exit stub version 2.0.0. Figures 3 to 5 depict the report for the `cfsan` plugin and Figures 6 to 7 contain the report for the `ptrsan` plugin.



Report ControlFlowSanitizationPlugin

Plugin description: Detects attacker-controlled jump targets.

Analyzed 'main.elf', with 'Sancus' enclave runtime. Ran for 0:00:03.088532 on 2024-06-01_19-03-28.

Enclave info: Address range is [Text: 0x6c64, 0x6d8d; Data: 0x200, 0x32b]

Summary: Found 1 unique WARNING issue; 1 unique CRITICAL issue.

Report summary

Severity	Reported issues
WARNING	• *Concrete ret target in non-executable memory* at 0x6d34
CRITICAL	• *Symbolic unconstrainted tainted jmp target* at 0x6cde

Report details (click to uncollapse)

✓ DEBUG ✓ INFO ✓ WARNING ✓ ERROR ✓ CRITICAL

Issues reported at 0x6d34 1 __sm_basic_enclave_ret_entry WARNING

Concrete ret target in non-executable memory

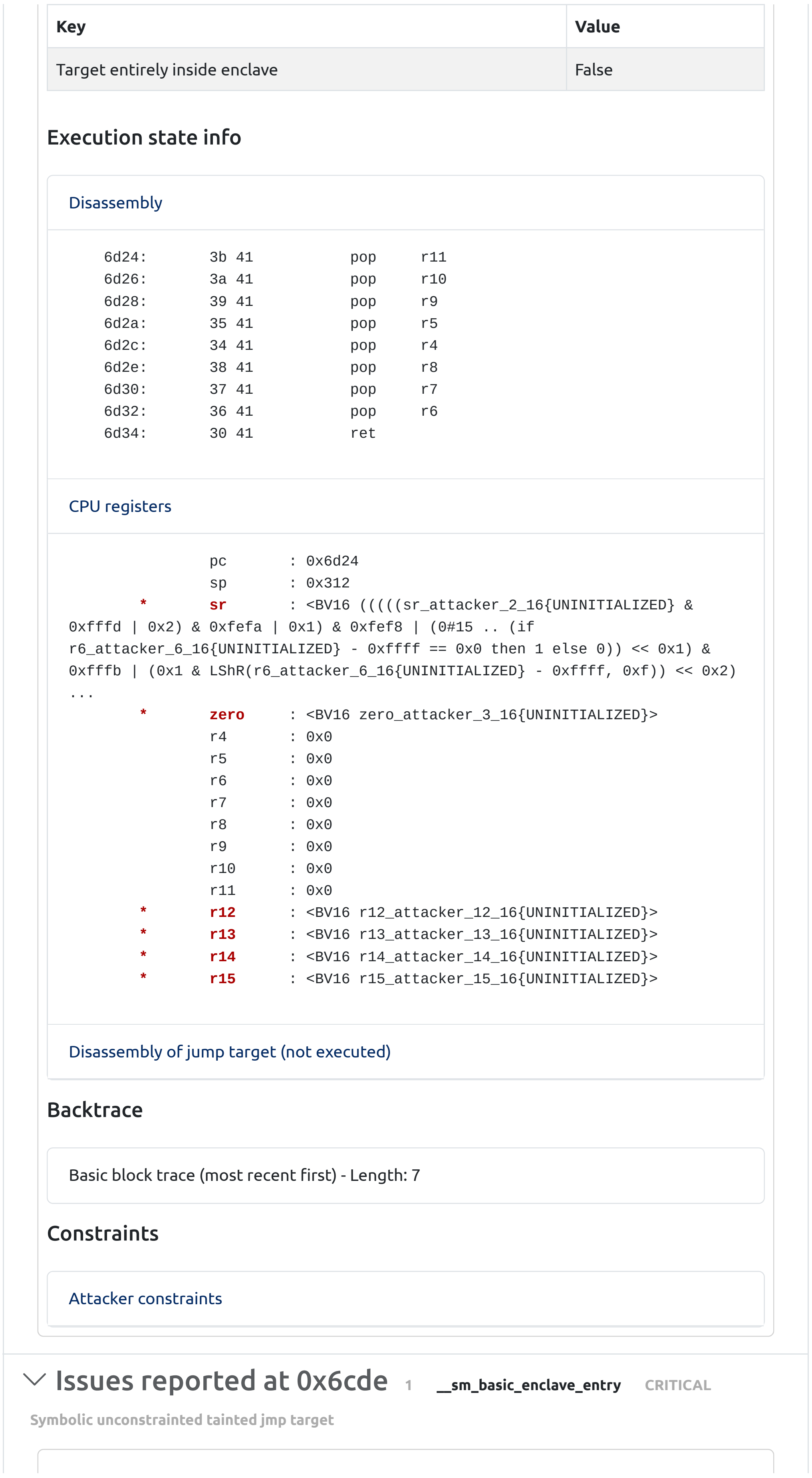
Concrete ret target in non-executable memory WARNING

IP=0x6d34

Plugin extra info

Key	Value
Target	0
Attacker tainted	False
Symbolic	False
Target range	[0x0, 0x0]

Figure 3. The `cfsan` report for the Sancus stubs v2.0.0 (page 1/3).



Key	Value
Target entirely inside enclave	False

Execution state info

Disassembly

```
    6d24:       3b 41           pop     r11
    6d26:       3a 41           pop     r10
    6d28:       39 41           pop     r9
    6d2a:       35 41           pop     r5
    6d2c:       34 41           pop     r4
    6d2e:       38 41           pop     r8
    6d30:       37 41           pop     r7
    6d32:       36 41           pop     r6
    6d34:       30 41           ret
```

CPU registers

```
                pc       : 0x6d24
                sp       : 0x312
        *       sr       : <BV16 (((((sr_attacker_2_16{UNINITIALIZED} &
0xfffd | 0x2) & 0xfefa | 0x1) & 0xfef8 | (0#15 .. (if
r6_attacker_6_16{UNINITIALIZED} - 0xffff == 0x0 then 1 else 0)) << 0x1) &
0xfffb | (0x1 & LShR(r6_attacker_6_16{UNINITIALIZED} - 0xffff, 0xf)) << 0x2)
...
        *       zero     : <BV16 zero_attacker_3_16{UNINITIALIZED}>
                r4       : 0x0
                r5       : 0x0
                r6       : 0x0
                r7       : 0x0
                r8       : 0x0
                r9       : 0x0
                r10      : 0x0
                r11      : 0x0
        *       r12      : <BV16 r12_attacker_12_16{UNINITIALIZED}>
        *       r13      : <BV16 r13_attacker_13_16{UNINITIALIZED}>
        *       r14      : <BV16 r14_attacker_14_16{UNINITIALIZED}>
        *       r15      : <BV16 r15_attacker_15_16{UNINITIALIZED}>
```

Disassembly of jump target (not executed)

Backtrace

Basic block trace (most recent first) - Length: 7

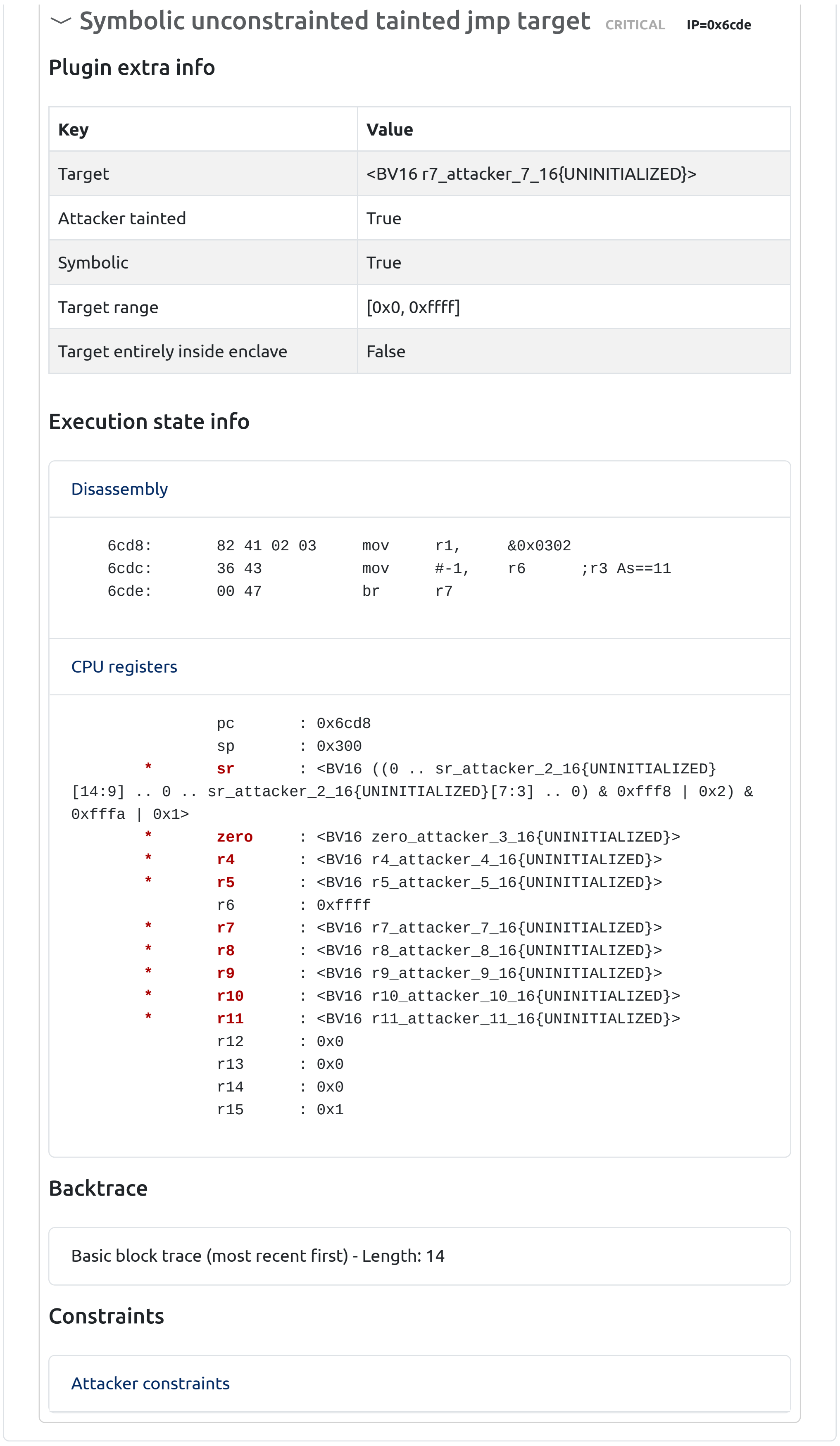
Constraints

Attacker constraints

Issues reported at 0x6cde 1 __sm_basic_enclave_entry CRITICAL

Symbolic unconstrainted tainted jmp target

Figure 4. The cfsan report for the Sancus stubs v2.0.0 (page 2/3).



﹀ Symbolic unconstrainted tainted jmp target CRITICAL IP=0x6cde

Plugin extra info

Key	Value
Target	<BV16 r7_attacker_7_16{UNINITIALIZED}>
Attacker tainted	True
Symbolic	True
Target range	[0x0, 0xffff]
Target entirely inside enclave	False

Execution state info

Disassembly

```
    6cd8:       82 41 02 03     mov     r1,     &0x0302
    6cdc:       36 43           mov     #-1,    r6      ;r3 As==11
    6cde:       00 47           br      r7
```

CPU registers

```
                pc       : 0x6cd8
                sp       : 0x300
        *       sr       : <BV16 ((0 .. sr_attacker_2_16{UNINITIALIZED}
[14:9] .. 0 .. sr_attacker_2_16{UNINITIALIZED}[7:3] .. 0) & 0xfff8 | 0x2) &
0xfffa | 0x1>
        *       zero     : <BV16 zero_attacker_3_16{UNINITIALIZED}>
        *       r4       : <BV16 r4_attacker_4_16{UNINITIALIZED}>
        *       r5       : <BV16 r5_attacker_5_16{UNINITIALIZED}>
                r6       : 0xffff
        *       r7       : <BV16 r7_attacker_7_16{UNINITIALIZED}>
        *       r8       : <BV16 r8_attacker_8_16{UNINITIALIZED}>
        *       r9       : <BV16 r9_attacker_9_16{UNINITIALIZED}>
        *       r10      : <BV16 r10_attacker_10_16{UNINITIALIZED}>
        *       r11      : <BV16 r11_attacker_11_16{UNINITIALIZED}>
                r12      : 0x0
                r13      : 0x0
                r14      : 0x0
                r15      : 0x1
```

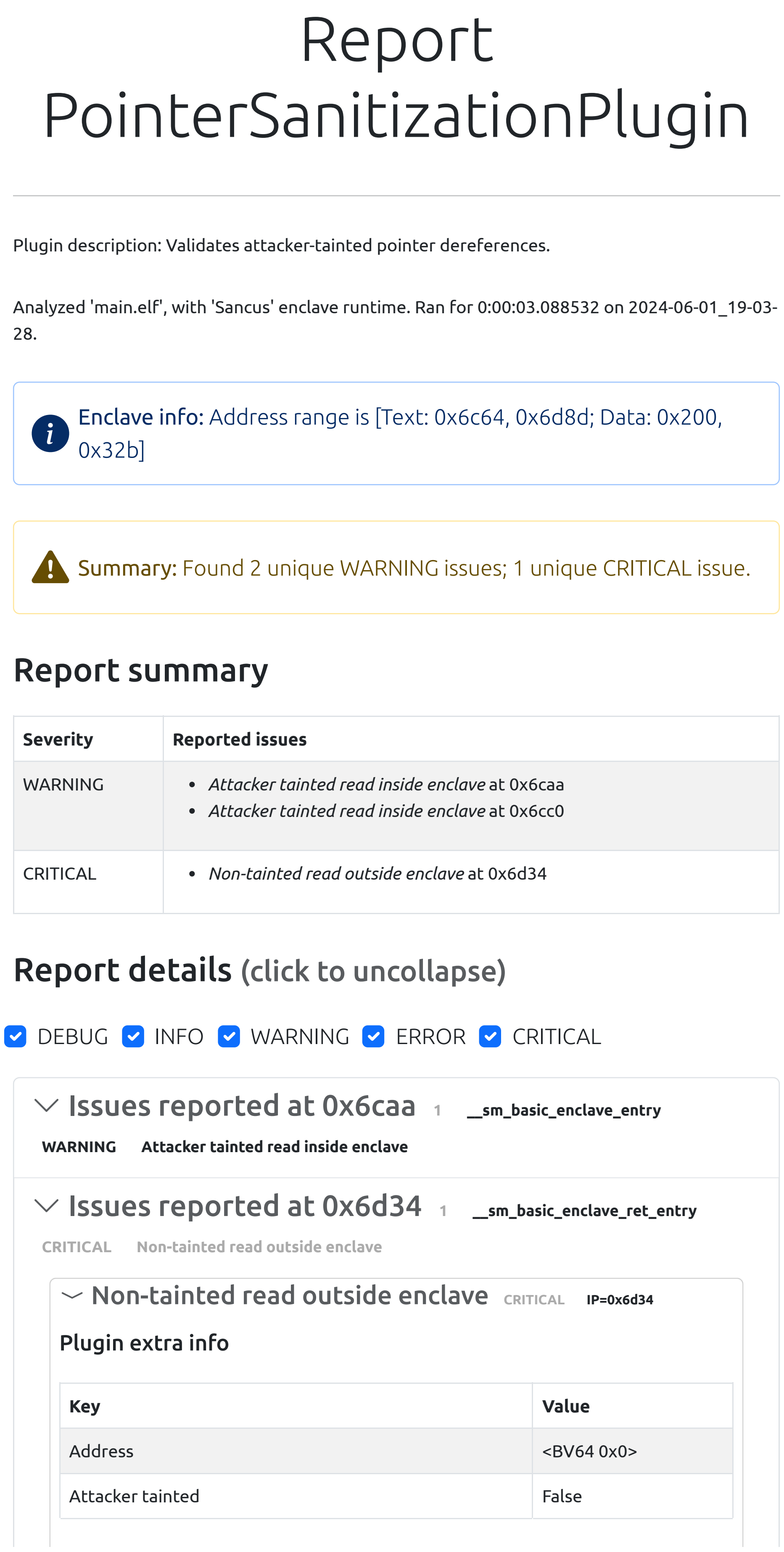
Backtrace

Basic block trace (most recent first) - Length: 14

Constraints

Attacker constraints

Figure 5. The `cfsan` report for the Sancus stubs v2.0.0 (page 3/3).



Report PointerSanitizationPlugin

Plugin description: Validates attacker-tainted pointer dereferences.

Analyzed 'main.elf', with 'Sancus' enclave runtime. Ran for 0:00:03.088532 on 2024-06-01_19-03-28.

Enclave info: Address range is [Text: 0x6c64, 0x6d8d; Data: 0x200, 0x32b]

Summary: Found 2 unique WARNING issues; 1 unique CRITICAL issue.

Report summary

Severity	Reported issues
WARNING	• *Attacker tainted read inside enclave* at 0x6caa • *Attacker tainted read inside enclave* at 0x6cc0
CRITICAL	• *Non-tainted read outside enclave* at 0x6d34

Report details (click to uncollapse)

✓ DEBUG ✓ INFO ✓ WARNING ✓ ERROR ✓ CRITICAL

Issues reported at 0x6caa 1 __sm_basic_enclave_entry

WARNING **Attacker tainted read inside enclave**

Issues reported at 0x6d34 1 __sm_basic_enclave_ret_entry

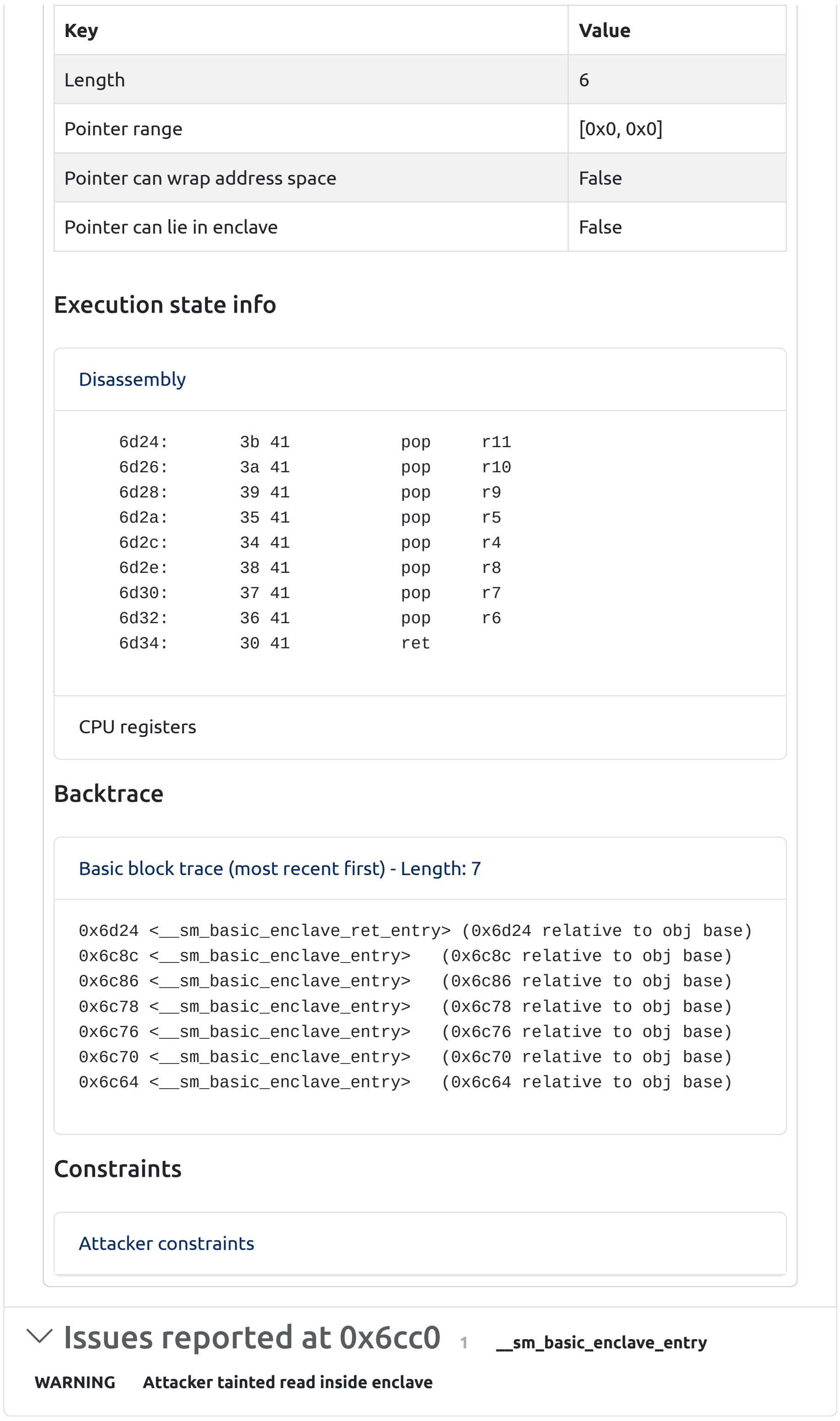
CRITICAL Non-tainted read outside enclave

Non-tainted read outside enclave CRITICAL IP=0x6d34

Plugin extra info

Key	Value
Address	<BV64 0x0>
Attacker tainted	False

Figure 6. The ptrsan report for the Sancus stubs v2.0.0 (page 1/2).



Key	Value
Length	6
Pointer range	[0x0, 0x0]
Pointer can wrap address space	False
Pointer can lie in enclave	False

Execution state info

Disassembly

```
    6d24:       3b 41           pop     r11
    6d26:       3a 41           pop     r10
    6d28:       39 41           pop     r9
    6d2a:       35 41           pop     r5
    6d2c:       34 41           pop     r4
    6d2e:       38 41           pop     r8
    6d30:       37 41           pop     r7
    6d32:       36 41           pop     r6
    6d34:       30 41           ret
```

CPU registers

Backtrace

Basic block trace (most recent first) - Length: 7

```
0x6d24 <__sm_basic_enclave_ret_entry> (0x6d24 relative to obj base)
0x6c8c <__sm_basic_enclave_entry>   (0x6c8c relative to obj base)
0x6c86 <__sm_basic_enclave_entry>   (0x6c86 relative to obj base)
0x6c78 <__sm_basic_enclave_entry>   (0x6c78 relative to obj base)
0x6c76 <__sm_basic_enclave_entry>   (0x6c76 relative to obj base)
0x6c70 <__sm_basic_enclave_entry>   (0x6c70 relative to obj base)
0x6c64 <__sm_basic_enclave_entry>   (0x6c64 relative to obj base)
```

Constraints

Attacker constraints

Issues reported at 0x6cc0 1 **__sm_basic_enclave_entry**

WARNING **Attacker tainted read inside enclave**

Figure 7. The `ptrsan` report for the Sancus stubs v2.0.0 (page 2/2).