Zellic

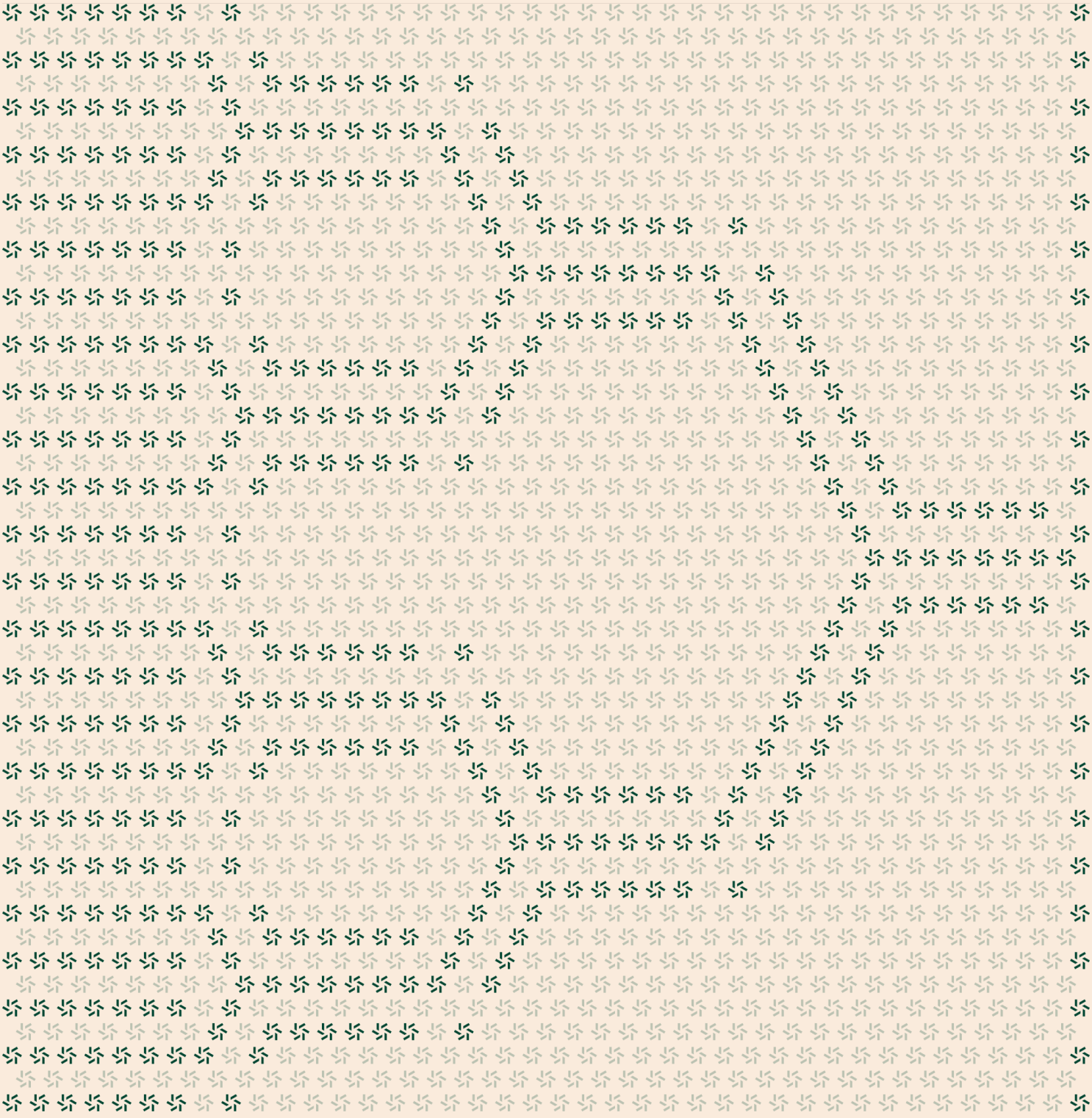
Prepared for
Uma Roy
Succinct Labs

Prepared by
Luna Tong
Zellic

December 4, 2024

SP1

Design Review



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic was approached by Succinct on November 31, 2024 to investigate the potential impact of some reserved memory in SP1 and the accompanying undefined behavior.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is SP1 in compliance with the RISC-V specification?
- What are the security impacts of the register-memory aliasing?
- Are there programs which rely on the undefined behavior but are nonetheless valid?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Correctness of any and all particular implementation details of the SP1 implementation. We assume the SP1 implementation is already audited by competent security professionals.
- Correctness of any and all RV32IM instructions in SP1. We focused on the particular instructions that were relevant to the questions posed to us by Succinct, mostly the load and store instructions.
- Soundness and/or completeness of any the ZK proofs or circuits. We assume these are already audited by competent security professionals.

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. During this assessment, we only considered SP1 from a design level overview. This was not a code audit.

1.4. Results

In summary, our opinion is that:

- Aliasing of registers and low memory (addresses 0x0–0x1f) is **NOT** a security vulnerability, as it does not affect valid programs, nor is it in conflict with the RV32IM specification;

- However, it could make it easier to develop exploits for programs that are **already** vulnerable;
- This implementation detail is not a design constraint or requirement;
- Therefore, as an **enhancement**, we recommend Succinct to consider adding a check to SP1's loads and store instructions to prohibit or ignore these operations to these low addresses. This enhancement should be viewed similarly to well-known mitigations like SMEP, SMAP, ASLR, or NX: not required for the normal operation of *safe* programs, but helpful for mitigating the weaknesses of *already-faulty* programs.

2. Introduction

Succinct is the developer of SP1. SP1 is a zkVM built on Plonky3 that implements the RV32IM (RISC-V 32-bit base with multiplication extensions) instruction set.

Rather than implement RV32IM exactly, SP1 implements a subset RV32I for proving efficiency. The limitations are documented here ↗, which we reproduce below:

- LW/SW memory access must be word aligned.
- LH/LHU/SH memory access must be half-word aligned.
- Memory access is only valid for addresses 0x20 through 0x78000000. Accessing addresses outside of this range will result in undefined behavior.
- The ECALL instruction is used for system calls and precompiles.

The limitation and undefined behavior we are particularly interested in is the invalid memory from address 0x0 to 0x1F. As an implementation detail, in SP1 each of these memory addresses is memory mapped to one of the 32 RISC-V registers. For instance, a write to memory at 0x5 would write to the register `x5`. The same goes for memory reads. Furthermore, writing values to memory at 0x0 would allow the program to set the value of the `x0`, which according to the RISC-V specification is supposed to be "hardwired to the constant 0". This functionality is undefined behavior according to Succinct.

2.1. Scope

The engagement involved a review of the following targets:

SP1 Crates

Type	Rust
Platform	RISC-V, Plonky3
Target	SP1
Repository	https://github.com/succinctlabs/sp1 ↗
Version	`15be73d3acae3718a0acc529ed566f6a448b1c0e`
Programs	`crates/core/executor` `crates/prover`

2.2. Project Overview

Zellic was contracted to perform a review for a total of 1 engineer-days. The assessment was conducted by one consultant over the course of two calendar days.

Contact Information

The following project manager was associated with the engagement:

Luna Tong
CEO
audit@zellic.io ↗

The following consultant was engaged to conduct the assessment:

Luna Tong
CEO
audit@zellic.io ↗

2.3. Project Timeline

The key dates of the engagement are detailed below.

December 2, 2024	Start of primary review period
December 3, 2024	End of primary review period

3. Discussion

3.1. The question at hand

On or around November 1, Succinct received a report from Mauro Toscano, 3MI, and LambdaClass claiming that it provides "a way to generate programs that verifies false statements", which seems to be "related to the mapping of registers to memory".

In response, we were posed with the following question by Succinct:

> Are there any class of programs that seem "benign" but actually have undefined behavior or can be exploited due to the fact that registers and memory are in the same address space?

In short, the answer is No. Programs which trigger undefined behavior are, by definition, invalid and therefore should not be considered. If Succinct or the SP1 ecosystem is interested in writing or supporting programs which rely on this behavior, it should be standardized in the SP1 specification.

But to answer this question more thoroughly and with greater nuance, we should consider the actual context SP1 is used in. Developers are proving their own applications, and they control the code and deployment. It is a single-tenant environment with only one program running at a time. Thus, the primary risks come from either: (1) unintentional vulnerabilities when handling untrusted input; or (2) intentionally malicious code trying to hide backdoors.

Scenario 1: Application-level memory corruption

In cases where programs handle untrusted input and contain memory safety issues, the ability to affect registers with writes to low memory could be a useful gadget for exploit development. Null pointer dereference (NPD) is a common but usually worthless bug to attackers because the null page is unmapped in modern userspaces. This gadget would allow attackers to upgrade a NPD to a register write. Combined with the fact that register 0 is used as a reserved zero register, this could cause serious issues for vulnerable programs.

However, this is still fundamentally a program vulnerability, not a VM issue. The program has already violated memory safety assumptions by allowing invalid or arbitrary memory access through bugs like buffer overflows or null pointer dereference. In memory safe languages like Rust, these kinds of accesses can only occur through the use of `unsafe` code or other safety failures. Thus, while SP1's current behavior could make exploitation easier, the root cause lies in the application's inherent memory safety issues. When there is memory corruption, all bets are off.

Nonetheless, while the root cause lies in the program's implementation, there is precedent for platforms implementing defensive mitigations. Modern hardware and operating systems include numerous security features like SMEP, SMAP, NX bits, and ASLR–none of which are necessary for memory-safe programs, yet provide protection against common vulnerability classes. Thus, we agree with these statements from the original report:

> It's also not true that you cannot do anything to prevent users from writing in places where it shouldn't...additional checks to avoid this kind of behaviour, even going further by disallowing IO from writing in critical parts of the memory.

> We suggest fixing it, or at least adding a further warning that it's not just undefined behaviour [...]

We recommend Succinct modify the implementation of SP1—while maintaining 0x0 through 0x1F as reserved memory—to simply prohibit or ignore writes to these addresses at the instruction handler level. For example, this could be implemented with a constraint that the address is >= 0x20. This would render unprovable the execution of programs that load/write in the aliased range.

We note that in our view, such a mitigation would be moreso an enhancement than a fix. This is analogous to how the absence of ASLR or DEP does not cause vulnerabilities, but their presence makes exploitation of existing vulnerabilities more challenging. It is not the platform's responsibility to ensure that application developers write correct programs. We disagree with this statement from the original report:

> Leaving this issue aside, it would be good to avoid errors that programs running in normal a machine can't have, or that can be avoided by simple checks that any OS have. Else we will have hackers simply attacking the programs themselves instead of the verifiers, and so breaking the soundness in an indirect way.

It cannot be assumed hackers will attack programs or not based on the absence or presence of any particular mitigation. Despite a plethora of mitigations on modern commercial platforms like iOS and Linux, sophisticated exploit chains are still developed and used in the wild ↗.

Scenario 2: Malicious code or backdoors

The original report argues that:

> Not having this in a general purpose VM, with projects not so rarely having over 800 dependencies, and with people doing custom patches to libraries for optimizations, makes it quite vulnerable to supply change attack. By having only one small dependency compromised all the system can be critically compromised.

The potential for hiding malicious behavior by accessing reserved memory regions is worth considering, but should be viewed in proper context.

First, this is not fundamentally different from the other myriad ways to write underhanded code. There are well-known contests for underhanded C ↗ and Solidity ↗ code. Attackers could draw inspiration from one of the many instances of prior art—from misleading names to unexpected compiler

optimizations or confusing proc macros. Adding special platform-level protections against one particular technique does not meaningfully improve security when so many other attack vectors exist. Any sufficiently motivated attacker can find alternative ways to implement hidden behavior.

Second, the responsibility to catch malicious behavior rests in code review, not the platform. In our industry, users generally distrust code that has not been audited. This fear is well-placed: even on Ethereum mainnet, countless users have been scammed through rug pulls placed in smart contracts. Despite this risk, the EVM is widely adopted. The problem is not with the EVM platform, but those particular contracts. Even if some projects have "over 800 dependencies", in general the responsibility to detect malicious code in applications is the responsibility of auditors, not the platform. We cannot expect a VM to prevent malicious programs from being run on top of it—this is impossible ↗, and "malicious" is not clearly definable.

Most importantly, if an attacker can insert malicious code, they already have full control over the program's behavior and do not need to rely on implementation quirks to achieve their goals. An attacker with the ability to modify source code can implement whatever behavior they desire using standard, well-defined platform features. The presence or absence of access to reserved memory regions does not meaningfully impact their capabilities.

As with the previous scenario, adding checks within the implementation of the load and store instructions to simply prohibit or ignore writes to this reserved region obviates this risk. We agree with the original reporters that this quirk *could* facilitate hiding malicious code. However, this scenario—the potential for deliberately malicious code—**in general** should not represent a meaningful security issue within the platform itself. There are countless ways to implement underhanded behavior in code, and preventing all such possibilities is neither practical nor the responsibility of the underlying platform. In Zellic's view, this is a matter of code review and trust, rather than platform security.

3.2. Standards Compliance

In this section, we discuss whether Succinct's design choices meaningfully affect its compliance with the RISC-V specification, specifically the RV32IM standard.

At a high level, we believe SP1 is generally in compliance both (1) in spirit and (2) in letter (except for the documented deviations ↗, namely memory access alignment).

In the original report, Mauro contends that:

> Additionally, since this kind of behaviour is not to be expected in a real RiscV machine, malicious code like this may be even easier to hide.

Zellic's view here is that this statement misunderstands what it means to be a "real RiscV [sic] machine". This term is misleading. RISC-V is a family of ISAs, not any one single ISA. More importantly, RISC-V is fundamentally created for customization and variation for the particular needs of the application (here, a zkVM). In our view, an implementation of RISC-V (SP1) that slightly deviates to better suit the underlying target technology (Plonky3) is no less "real" of a RISC-V machine.

If RISC-V was a "commercial off-the-shelf" like x86 or ARM primarily intended to be used as-is

without modification, the statement would raise an important concern. But unlike x86 or ARM, RISC-V is an open ISA meant for customization and specialization. Indeed, according to the RISC-V specification's ↗ introduction:

> RISC-V has been designed to support extensive customization and specialization. The base integer ISA can be extended with one or more optional instruction-set extensions, but the base integer instructions cannot be redefined. ... The base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems (with additional supervisor-level operations), and so provides a convenient ISA and software toolchain "skeleton" around which more customized processor ISAs can be built.

In other words, while off-the-shelf RISC-V is a convenient target ISA, the primary purpose of RISC-V is to act as a "skeleton" on top of which ISA developers can further customize, modify, and specialize. Crucially, the existing software toolchain surrounding RISC-V is a starting point, not the destination. As for SP1, the base integer instructions are all implemented and fully supported, clearly demonstrating alignment with the RV32I standard.

Moreover, the first page of the RISC-V specification's introduction defines the key design goals:

> - An ISA that avoids "over-architecting" for a particular microarchitecture style...or implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these.

> - An ISA supporting extensive user-level ISA extensions and specialized variants.

The goal of RISC-V is not to enshrine some particular ISA that must be exactly adhered, but to provide a reasonable base for efficient implementation. Emphasis lies not on conformance, but on efficiency and pragmatism. Implementations may need to *specialize* or *vary* the ISA for efficiency. When implementations of RISC-V vary from the standard in minor ways for good reason, it is not a failure of the implementation but a consequence of what appears to be the general philosophy of RISC-V.

Considering ZK circuits are a distinctive implementation technology harboring unique challenges and constraints, one would reasonably expect RISC-V implementations for ZK circuits to potentially have variations or specialization for efficiency reasons. The limitations SP1 imposes on its ISA are indeed for efficiency reasons; according to the SP1 book ↗, they are "implementation details that make it more suitable for proving". Thus, in our opinion SP1 is very much a "real" RISC-V machine, in that it aligns with the overall design principles underlying RISC-V.

We also disagree with the following statement ↗:

> If, as an industry, we accept deviations from the RISC-V specification, we risk introducing subtle and complex bugs.

First, we do not believe the introduction of the reserved memory regions or the undefined behavior violate the RISC-V specification. We discuss the reasons why we believe this to be the case in the following section.

Second, we believe that the deviations do not harbor a significant risk of introducing "subtle and complex bugs". In practice, does requiring access alignment and defining invalid memory regions realistically hinder the usability of the SP1 ISA and execution environment? Zellic's position here is that, for reasonable developers and users, it does not, for the reasons aforementioned in the first section of this report.

Third, we disagree with the statement's overall absolutist stance on RISC-V; it is not so black and white. Again, it is reasonable for implementations to have certain deviations for efficiency reasons provided that the deviations are well-documented and well-supported in the software toolchain. It is reasonable to expect developers to familiarize themselves with the documentation of the platform they are building on top of. Moreover, it is not clear what "deviation" from spec the statement is referring to. Considering that the introduction of reserved memory regions is not noncompliant, the other main deviation is requiring memory access to be aligned. However, this deviation is not drastic, appears reasonable, and the original report did not raise alignment as an issue.

3.3. Reserved memory regions are part of the RISC-V specification

In Section 2.6, Load and Store Instructions, the RISC-V standard states that "the execution environment will define what portions of the address space are legal to access". Nowhere in the RISC-V standard does it prohibit implementers from specifying certain things as undefined behavior.

Many hardware platforms, especially embedded systems, define special memory regions which are either reserved (e.g., as SMM memory on x86, ISA DMA at 15-16MB), invalid (e.g., gaps in physical memory space), or have other special semantics (e.g., MMIO like GPIO registers, timer registers, ADC/DAC control registers, DMA control registers, mailslots, PCIe BARs, ROM/firmware regions, etc).

SP1 specifies that memory addresses from 0x20 to 0x78000000 are valid, and access to other memory addresses constitutes undefined behavior (UB). Compared to the memory maps of many real-world systems, SP1's memory map is no more complicated or unusual, and we see no problem with this. Furthermore, we find that it is compliant with the RISC-V specification.

In real-world platforms, access to illegal or special memory regions can trigger unpredictable behavior. For example, writes to DMA regions could corrupt memory contents; writes to power management regions could cause the machine to simply turn off. This is completely reasonable. If a privileged program were to make these writes unintentionally, it is the program that is faulty, not the platform. Thus, while SP1's aliasing of registers to low memory is an interesting implementation quirk, for any programs that suffer from this quirk, it is a problem with the program, not the SP1 VM. The platform defines these memory-mapped register regions as invalid memory, and programs

should simply not access them. Thus, we disagree with this statement from the original report:

> While it's true that you cannot avoid malicious programs, no program should be able to break the ISA of the VM. If you manage to rewire R0 to another value for example, it's quite a change in the definition of what the programs can do, and it can be quite bad.

This statement is concerned with "the definition of what the programs can do". But by definition, undefined behavior allows the implementation to do *anything*—including crash, produce incorrect results, or cause the machine to catch fire. By definition, UB is allowed to "break the ISA of the VM", even by enabling hidden processor modes ↗ for example.

Nevertheless, we also believe that UB is also generally undesirable in ISAs. Undefined behavior can lead to "pseudo-standards", when a particular implementation becomes the *de facto* standard due to applications being developed that rely on UB. The situation becomes confusing and error-prone for developers and implementers. To prevent this, we recommend that Succinct implement the mitigations mentioned elsewhere in the report, and standardize the new behavior in the SP1 documentation rather than leaving it as UB.

3.4. Other Findings

We reviewed the SP1 implementation as of commit 15be73d3 ↗. We noticed the following findings, which are all technically deviations from the RV32IM spec but are inconsequential. They are all essentially QA-level issues, which Succinct acknowledged:

- `FENCE` instruction triggers a "not implemented" fault, rather than acting as a NOP
- `WFI` instruction triggers a "not implemented" fault, rather than acting as a NOP
- `MRET` is not implemented
- CSR related instructions are not implemented

4. Conclusion

SP1's implementation of aliasing registers with low memory addresses (0x0-0x1f) is an interesting implementation decision. Technically, reserved memory ranges are not incompatible with the RISC-V specification, which explicitly provides affordances for implementers to define them, nor are the effects of undefined behavior. Nonetheless, while this is not a security vulnerability in itself, we recommend implementing instruction-level checks to ignore operations to these low memory addresses to strengthen SP1's security posture and reduce potential attack surface for vulnerable programs. Furthermore, we recommend Succinct document and standardize this, rather than leave it as undefined behavior.

4.1. Disclaimer

This report represents the opinion of Zellic. It does not represent the opinions of any other party including any of the parties referenced in the report. This report was commissioned with the support of Succinct.

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For our findings, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.