Validating Automatic Variable Initialisation

A static binary analysis approach to compiler code generation verification

Reference 26-04-2682-REP
Version 1.1
Date 2026-04-27

Quarkslab

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

Legal notice

This report reflects the work and results obtained within the specified scope and duration of the project as agreed between Quarkslab and the other parties.

1. Project information

1.1. Document history

Version	Date	Details	Authors
1.0	2026-04-01	Document creation and initial drafting	Francesco Cagnin
1.1	2026-04-27	Document review and validation	Francesco Cagnin

1.2. Contacts

1.2.1. Quarkslab

Name	Role	Email
Frédéric Raynal	CEO	fraynal@quarkslab.com
Pauline Sauder	Project Manager	psauder@quarkslab.com
Francesco Cagnin	R&D Engineer	fcagnin@quarkslab.com
Samuel Hangouët	R&D Engineer	shangouet@quarkslab.com
Nicolas Surbayrole	R&D Engineer	nsurbayrole@quarkslab.com

1.2.2. OSTIF

Name	Role	Email
Derek Zimmer	Executive Director	derek@ostif.org
Amir Montazery	Managing Director	amir@ostif.org
Helen Woeste	Operations, Communications and Community	helen@ostif.org
Tom Welter	Project Manager	tom@ostif.org

1.2.3. LLVM Foundation

Name	Role	Email
Kristof Beyls	Director	kristof.beyls@arm.com

Contents

2. Executive summary

This chapter provides an overview of the project, including the necessary background, scope, and a summary of the work carried out and the results obtained. A glossary of acronyms and binary-analysis terms is provided after the appendix.

2.1. Context

BOLT[1] is a post-link optimiser that rewrites compiled applications to improve their performance. Originally developed by researchers at Meta, the tool has been fully open-sourced and is now part of the LLVM project.[2]

In 2024, Kristof Beyls at Arm developed a prototype static binary analyser, built on top of BOLT, to validate compiler code generation for security-related features.[3-4] This tool operated directly at the binary level[5] to "verify that a given hardening feature has been applied correctly across the whole binary," by checking whether the compiled program satisfies the security properties the feature is intended to provide. This new approach to verification, aimed at uncovering compiler bugs, was intended to complement regression and unit tests, which validate code generation only for a few test cases. The decision of using BOLT was motivated by: the rich set of features for binary analysis it provides[6]; its broad support for binaries across different compilers and architectures; and its expected long-term maintainability.

The first scanner implemented for this binary analyser targeted the "pac-ret" hardening scheme for AArch64,[7] associated to the compiler flag `-mbranch-protection=pac-ret` for both GCC[8] and LLVM Clang.[9] The scanner seeks to detect unsafe instruction patterns that could weaken the security guarantees expected from code that uses Pointer Authentication correctly. A second scanner that focuses instead on code generation mechanisms that prevent Stack Clash[10] attacks (compiler flag `-fstack-clash-protection`) is also under development.

Given the inherent limitations of static analysis, scanners that rely on it typically have to trade off soundness for completeness. Reporting every potential issue increases the number of false positives, generating noise that costs time to sort through; reporting only fully verified violations raises the risk of false negatives, possibly resulting in genuine misses of code generation bugs. The former approach is generally preferred over the latter.

[1] https://research.facebook.com/publications/bolt-a-practical-binary-optimizer-for-data-centers-and-beyond/
[2] https://github.com/llvm/llvm-project/blob/main/bolt/README.md
[3] https://discourse.llvm.org/t/rfc-bolt-based-binary-analysis-tool-to-verify-correctness-of-security-hardening/78148
[4] https://github.com/llvm/llvm-project/blob/main/bolt/docs/BinaryAnalysis.md
[5] Source-level analysis cannot be used to validate the compiler's output.
[6] For example, control-flow graph reconstruction and data-flow and reaching definition analyses.
[7] https://github.com/llvm/llvm-project/commit/850b49297615a613ac83adca2c9cf823a4b8ef95
[8] https://gcc.gnu.org
[9] https://clang.llvm.org
[10] https://blog.qualys.com/vulnerabilities-threat-research/2017/06/19/the-stack-clash

2.2. Objectives

The Open Source Technology Improvement Fund[11] (OSTIF) is a nonprofit dedicated to securing open source software. With the goal of further developing the BOLT-based binary analyser in LLVM, OSTIF commissioned Quarkslab to extend it to support additional compiler flags for security hardening. The initial mandate was intentionally broad, leaving it to Quarkslab to identify which code generation schemes would be appropriate to tackle, balancing relevance, complexity, and the resources available for the project. Following a preliminary assessment, Quarkslab was then expected to propose one or more hardening features to work on and, upon agreement with OSTIF, implement corresponding scanners for the binary analyser.

2.3. Work summary

Candidate compiler hardening options were first reviewed, and `-ftrivial-auto-var-init` was selected as the most suitable target. The flag is implemented by both GCC and Clang, is relevant to current hardening practice, and has a clear binary-level property to check: automatic variables should be initialised before use.

Two complementary approaches were then developed for validating the flag at the binary level:

- a load-oracle approach, which uses stack loads as evidence of automatic variables residing on the stack; and
- a store-witness approach, which checks whether the bytes read by each stack load were written on every relevant control-flow path before the load.

These ideas were implemented in a new BOLT-based scanner. The scanner recovers direct and derived stack accesses, recognises common block initialisation patterns, verifies initialisation within and across functions, and emits both diagnostic reports and limitation reports to make triage explicit.

The scanner was evaluated on GNU coreutils 9.5 across GCC and Clang x86-64 builds, at several optimisation levels, with and without `-ftrivial-auto-var-init=zero`. The evaluation was intended to carry the implementation forward and identify its remaining limits, not to assess coreutils as a security target.

2.4. Conclusion and next steps

The evaluation shows that a BOLT-based scanner can capture some binary-level evidence for `-ftrivial-auto-var-init`, limited to the stack accesses it can recover. On the coreutils corpus, enabling automatic initialisation produced the expected drop in missing-initialisation diagnostics, but the remaining reports also confirm the precision and coverage limits inherent to static analysis. Larger and more complex targets, especially C++ programs, are likely to expose further cases and generate substantially more reports, reducing the scanner's usefulness as triage

[11] https://ostif.org

becomes considerably more cumbersome. The same observations apply when the scanner is used to detect genuine uninitialised stack uses rather than to validate code generation.

To develop the implementation further, possible next steps include supporting non-unique cross-basic-block stores, improving pointer-to-stack resolution, and, where possible, enhancing BOLT's control-flow graph reconstruction and frame recovery. Before doing so, however, the scanner should be tested on more demanding targets to build a complete picture of the main sources of remaining false positives, so that initial improvements can focus on those with the greatest impact. Pruning infeasible CFG paths may be one such improvement.

Overall, making the approach more effective in practice will likely also require auxiliary techniques, such as DWARF-assisted variable recovery, differential compilation, and targeted symbolic execution. A key strategic decision is therefore whether future work should continue to target stripped binaries or whether relying on richer build-time information is acceptable.

Either way, some of the mechanisms built for this scanner, including propagation of non-returning calls and resolution of path-constrained stack access, should be reusable for validating other compiler hardening flags or supporting other BOLT use cases.

3. Analysis of candidate compiler flags

This chapter presents our exploratory work to review existing compiler options for security hardening and ultimately select one for implementing a static binary scanner.

3.1. Collecting relevant options

To complement our prior knowledge on the topic and bring it up to date, we began our investigation by surveying the security hardening options provided by two major compilers, GCC and Clang. To navigate the wide range of common and lesser-known flags for tweaking code generation, we examined both the compilers' online documentation and the "Compiler Options Hardening Guide for C and C++"[12] from the Open Source Security Foundation (OpenSSF) Best Practices Working Group. Eventually, we compiled a reasonably comprehensive list of relevant flags, summarising for each option:

- its availability by architecture (limited to x86-64 and AArch64) and compiler (GCC and Clang);
- its estimated adoption (discussed below);
- the mitigations it provides and the type of attacks it helps prevent (e.g. "Control Flow Integrity" and "CF hijacking");
- a short outline on what the option actually does; and
- a surface-level assessment on the feasibility of validating its implementation through static analysis.

The full results of our study are deferred to Appendix A; this chapter only provides the necessary overview to explain how we narrowed the options down to one.

3.1.1. Evaluating adoption

One of the criteria we planned to use to determine whether a compiler flag might be worth investigating more was its actual degree of adoption. To evaluate it summarily, we sought to collect the default options used by some Linux distributions to compile packages, and quickly realised that recovering the exact flags is not straightforward, for several reasons. First, each distribution has its own ways to specify flags for compiling packages: the configuration files or mechanisms change from distro to distro, and also between versions of the same distribution over time. Second, some flags are specific to the target architecture, so a package compiled for x86-64 often uses slightly different options than the same package built for AArch64. Third, compilers are configured[13] with a set of flags that are enabled even when not explicitly selected by the user; for example, GCC configured with `--enable-default-ssp` (very common nowadays) inserts stack canaries by default into any program it compiles. These preset options develop with time, so that successive versions of the same compiler may implicitly enable distinct flags. Moreover, even using the same compiler version is not exempt from default flags changes, since

[12] https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.html

[13] https://gcc.gnu.org/install/configure.html

each distribution may build the shipped compiler with a different configuration, e.g. `--disable-default-ssp`. Finally, each package is free to provide its own set of flags to use, often buried deep in their build systems. With all these variables involved, the only reliable way to determine which flags are actually used when compiling software is to build it in verbose mode and inspect the compiler commands used for each file.

Our adoption study (Table 6 in Appendix A) is then to be considered only a rough estimate of how widely flags are used in practice. It assigned to each flag an approximate score that weights in several factors, including whether the option is:

- recommended by OpenSSF;
- included in GCC's `-fhardened` set[14];
- used by default to compile packages in recent versions of Debian, Ubuntu, and Arch Linux; and
- used by default to compile the Linux kernel.

3.2. Selecting a target flag

Once the options were catalogued, we moved on to filtering them. The first pass discarded those considered out of project scope, which comprised several categories:

- flags for the linker, for example `-Wl,-z,relro,-z,now`;
- flags not intended for production, such as most of the `-fsanitize` family;
- flags for which scanners were already under development, i.e. `-mbranch-protection=pac-ret` and `-fstack-clash-protection`;
- flags that do not insert any identifiable binary-level construct to validate, including `-fno-strict-aliasing`, `-fno-strict-overflow`, `-fstrict-flex-arrays=3`, and `-fno-delete-null-pointer-checks`;
- flags that only enable hardware features, such as `-fcf-protection=return` and `-mbranch-protection=gcs`; and
- flags whose validation reduces to a binary metadata check, like `-fPIE`, `-fPIC`, and `-mshstk`.

The second pass aimed at retaining options with wider adoption or at least currently recommended by OpenSSF, eliminating those that were:

- superseded in practice, like `-D_FORTIFY_SOURCE=2` (in our observations fairly phased out in favour of `-D_FORTIFY_SOURCE=3`) and `-fvtable-verify=std` (rarely used);
- relatively niche, such as `-mretpoline`, `-mindirect-branch` and `-mfunction-return` (only for the kernel) and `-mspeculative-load-hardening`; and
- much less prevalent than the others, i.e. `-fsanitize=cfi` and `-fsanitize=shadow-call-stack`.

The options left after the first two screening passes are listed in Table 1.

[14] https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html#index-fhardened

Option	Overview	SBA feasibility	Adopt.
`-D_FORTIFY_SOURCE=3`	Replace some glibc functions (e.g. `memcpy`, `strcpy`) with variants adding boundary checks on buffer sizes at both compile time and run time.	Check that unsafe functions are used only in a context that can be statically proven to be safe. Dynamic object sizes cannot be evaluated statically, so whether a non-checked call was actually safe cannot always be confirmed: high risk of false positives.	High
`-D_GLIBCXX_ASSERTIONS`	Add run-time checks to some libstdc++ functions (for example to abort on invalid array access), which are often inlined.	Detecting the inserted assertions should be easy as each failed check calls an abort helper. Verifying coverage would however require enumerating every bounds-checked libstdc++ access, which after inlining cannot be distinguished from ordinary pointer arithmetic.	Medium
`-fstack-protector-strong`	Insert canaries in functions with local arrays, address-taken variables, calls to alloca, and more. Rearrange local variables so that arrays are closer to the canary.	Check that functions matching the `-strong` criteria have a canary check before returning. Requires identifying which functions have arrays, address-taken variables, or calls to alloca at the binary level: difficult without debugging data.	High
`-mbranch-protection=bti`	Add BTI to function entries and other indirect-branch targets.	Check that every indirect-branch target (function entry or jump-table case) starts with BTI. Enumerating these targets has high false positive risk on code that is never indirectly called in practice.	High

Option	Overview	SBA feasibility	Adopt.
`-fcf-protection=branch`	Enable CET Indirect Branch Tracking (IBT). Add ENDBR64 to function entries and other indirect-branch targets.	Check that every indirect-branch target begins with ENDBR64. Enumerating these targets has high false positive risk on code that is never indirectly called in practice.	High
`-fzero-call-used-regs=used`	Zero call-used registers at function return.	Check that all registers used by the function are zeroed before returning.	Low
`-ftrivial-auto-var-init`	Add pattern or zero initialisation to automatic variables that would otherwise remain uninitialised.	Check that all automatic variables are actually initialised before first use. Requires identifying the variable locations (on the stack and in registers), and checking that each location is written at least once before it is read. Risk of false positives if correct initialisation is difficult to verify, for example when initialisation happens in an opaque callee.	Medium
`-fzero-init-padding-bits=all`	Guarantee zero initialisation of padding bits in variable initialisers (structures, unions, arrays of such).	Check that padding bits in structures and unions are zeroed. Difficult or impossible without debugging data to identify padding regions.	Low
`-fsanitize=safe-stack`	Split the stack into a safe stack (return addresses, spilled registers, safe local variables) and an unsafe stack (to keep all the rest). Unsafe variables are moved to a separate allocation so overflows cannot corrupt return addresses.	Check that the stacks contain the variables they are supposed to contain, which is difficult or impossible without debugging data.	Low

Table 1: Flags passing the first rounds of selection.

The third and last pass filtered the remaining flags mostly by their estimated difficulty of validation through static binary analysis. The only candidate that appeared considerably easier to validate than the others was `-fzero-call-used-regs=used`; however, its low adoption and its apparent triviality pushed us to drop it in favour of more fitting targets for the project.

`-fzero-init-padding-bits=all` and `-fsanitize=safe-stack` were eliminated for their low adoption (the former is GCC-only, the latter Clang-only) and their low feasibility.

`-fstack-protector-strong` was discarded for two reasons: the option has been around long enough that compilers are likely to have the implementation correct by now; and the risk of false positives was deemed high, since the scanner would have to reconstruct the compiler's decision to include or omit the canary for each function.

`-D_FORTIFY_SOURCE=3` was ruled out on similar grounds, as validating it would require checking that the unsafe function calls that the compiler decided to not fortify can actually be proven to only operate in a safe context.

`-D_GLIBCXX_ASSERTIONS` was also rejected for low feasibility. To decide that an assertion is missing, a scanner would have to enumerate every libstdc++ call site that should have been checked (e.g. `std::vector::operator[]`); but once inlined, without information on C++ container types, these sites are indistinguishable from ordinary pointer arithmetic.

`-mbranch-protection=bti` and `-fcf-protection=branch`, basically the same feature for AArch64 and x86-64, were ruled out together, since establishing where landing pads should be placed requires reconstructing the set of indirect-branch targets across the whole binary. This problem has no sound static solution, with a high false-positive risk for functions that are never indirectly called in practice.

This left `-ftrivial-auto-var-init` as the most appealing candidate. It is implemented by both GCC and Clang, available for x86-64 and AArch64, recommended by OpenSSF, included in GCC's `-fhardened`, and part of the default Linux kernel hardened configuration. Moreover, the automatic initialisation introduced by this flag has been integrated into the C++26 standard.[15] At this exploratory stage this option seemed challenging to verify, yet possibly more tractable than most of the alternatives. For all these reasons, we eventually selected it as the focus of the project.

The rest of the report examines how thoroughly the implementation of this flag can be validated through static binary analysis.

[15] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2795r5.html

4. Analysis of `-ftrivial-auto-var-init`

This chapter examines the `-ftrivial-auto-var-init` compiler flag in detail. It first presents the option's scope, goals, and guarantees. It then explores how automatic initialisation could be validated at the binary level through two complementary techniques: a load-oracle approach for locating stack variables, and a store-witness approach for verifying their initialisation. To support both, it introduces a shared taxonomy of stack accesses before discussing the main limitations and special cases of each. Readers unfamiliar with the compiler and binary-analysis terminology used below may refer to the glossary.

4.1. Automatic variable initialisation

The `-ftrivial-auto-var-init` flag instructs the compiler to insert automatic initialisation for eligible automatic variables that would otherwise be left uninitialised. The covered set broadly comprises block-scope locals that are not declared static, thread local, or extern; other objects such as heap allocations and globals are not affected by this flag.

The guarantee is that the first use of any qualifying variable always sees an initialised value. The goal is to prevent data leaks or other exploitation of uninitialised memory. A program that uses an uninitialised variable generally exhibits undefined behaviour,[16] and may expose a previous value that the compiler decided to store in the same location that happens to be used by that variable.[17]

Most automatic variables end up stored on the stack, but depending on the optimisation level, the variable size, and other factors, the compiler may decide to store them directly in registers instead. Both cases are subject to automatic initialisation.

The value inserted through automatic initialisation is currently configured by selecting one of two possible modes. By specifying `-ftrivial-auto-var-init=pattern`, the compiler uses a predefined byte pattern, typically based on repetitions of the byte 0xAA. With `-ftrivial-auto-var-init=zero`, the inserted value consists of repeated null bytes. The pattern technique is recommended because more likely to transform missed initialisations into actual crashes with recognisable values, whereas zero-initialisation preserves semantics more often.[18] Nevertheless, the latter remains available for cases where pattern initialisation is considered too costly. Some implementations also provide ways (e.g. attributes) to skip selected variables or to cap automatic initialisation by size or count.

The flag only requires that initialisation is inserted before the variable's first use, leaving the compiler ample room to decide whether and how to emit it. For example, under the "as-if rule" initialisation may be omitted entirely as long as it does not change the observable

[16] Starting with C++26, many reads from uninitialised automatic objects have erroneous behaviour instead (cf. previous footnote).

[17] https://isocpp.org/files/papers/P2723R0.html

[18] https://lists.llvm.org/pipermail/cfe-dev/2018-November/060172.html

behaviour of the program. When initialisation is emitted, the compiler may choose between several architecture-dependent code patterns.

4.2. Verifying initialisation at the binary level

A scanner operating at the binary level to validate the implementation of `-ftrivial-auto-var-init` must be able to solve at least two separate problems: inferring automatic variables from machine code, and determining whether they have been initialised before their first use.

When compiler-generated debugging information is available (e.g. DWARF), locating variables should become relatively straightforward, since debug data records their location and size within the compiled code. Relying on this would be reasonable if we confined the scanner solely to validating code generation, given that the tester can generate the debugging data. However, the same scanner can also be repurposed to find genuine uninitialised-use bugs in programs not compiled with `-ftrivial-auto-var-init`, where debug information is often unavailable. To keep this secondary use case open, the rest of this chapter develops the analysis without relying on such data, though it could still supplement the scanner when available.

4.2.1. Detecting variables

Consider the simplest scenario of automatic initialisation, compiled with `-ftrivial-auto-var-init=zero` (and without optimisations, for the sake of discussion):

```c
int f1() {
    int a;
    return a;   // uninitialised use of a
}
```

```asm
; Clang AArch64 -O0 -ftrivial-auto-var-init=zero
f1:
        sub     sp, sp, #16         ; allocate a stack frame of 16 bytes
        str     wzr, [sp, #12]      ; write 4 null bytes at sp+12 (auto-
        initialisation of a)
        ldr     w0, [sp, #12]       ; load 4 bytes from sp+12 (access to `a`)
        add     sp, sp, #16         ; deallocate the stack frame
        ret
```

The object `a` is an automatic variable that the compiler placed somewhere on the stack. The stack frame of the compiled function is 16 bytes in size, even though it only contains a single variable of size 4 bytes; the remaining 12 bytes are unused (allocated only to keep the stack aligned for architectural constraints).

The automatic initialisation inserted by `-ftrivial-auto-var-init` (the `str` instruction in the example above) does not blindly initialise the whole stack frame, but only the parts allocated to variables, and only those that the compiler determined to be otherwise uninitialised. In principle,

zero-initialising each stack frame on entry would be a simpler and equally safe solution, but the runtime overhead of doing so is usually prohibitive.

Once the source program is compiled into machine code, information about variables, including their location and size, is lost. At source level, the definition of `a` is evident; at binary level, no explicit information remains about it. Without additional debugging data, operating solely on machine code provides no direct means to reconstruct that the original function had a single variable and which stack location it ended up occupying.

At the same time, this example demonstrates an indirect means that can be used to determine `a`'s presence on the stack: using the `ldr` instruction that accesses it as an oracle. This approach provides an imperfect solution to detecting automatic variables on the stack, and is discussed in depth later in the chapter.

The compiler might also decide to keep automatic variables in registers, which the load-oracle technique cannot detect. Locating such variables would require a separate analysis with its own set of challenges, which we did not explore in this study: the initialisation criterion below and the rest of the analysis focus exclusively on stack-resident variables. In practice the limitation is modest, since the variables most exposed to uninitialised-use issues, e.g. arrays, are more likely to end up on the stack.

4.2.2. Detecting initialisation

Once a stack variable has been identified, the next step is to determine whether it has been initialised. This can be reframed as checking whether the first access to the corresponding stack location is a store rather than a load, since the initial store is the one that initialises the variable. If a load occurs before it, then the variable is being used uninitialised, invalidating the guarantee that `-ftrivial-auto-var-init` should provide. We call this the store-witness approach: each store witnesses the initialisation of the bytes it writes.

More precisely, a load of n bytes is considered initialised only if, on every feasible control-flow graph path from the function entry to the load, each of those bytes has been written by at least one preceding store. Partial coverage does not suffice: initialising 7 of 8 bytes still leaves one byte carrying whatever value previously occupied the same location. The quality of the result therefore depends on how accurately the analysis can determine which CFG paths and stores are actually relevant.

The variable's initialisation may have been provided by the programmer or automatically by the compiler; at this level, it is not possible to distinguish between the two. However, this distinction is of little significance, since the property being verified is simply that every variable is initialised. Moreover, the stored value itself is never inspected; only the fact that the bytes are written is checked.

A buggy compiler emitting an initialising store whose source still held leaked content (e.g. a register carrying a prior variable's value) would therefore pass the check, even though this is exactly the kind of leak `-ftrivial-auto-var-init` is meant to prevent. Recognising such issues is not possible at the binary level alone. Even if a data-flow analysis could trace the origin of each stored byte, it would still be unable to determine whether its presence there was intended.

In fact, a deliberate `int b = a;` and a code-generation bug that leaks `a`'s value into `b`'s slot may produce the same machine instructions; telling them apart would require knowing the compiler's intent for each store, which even debug information does not record.

4.3. Taxonomy of stack accesses

Both the load-oracle and store-witness approaches depend on recognising stack accesses. At this level, loads and stores have the same structure: each access touches a range of bytes within the stack frame, and the analysis must recover that range from machine code. We therefore adopt a single taxonomy for stack accesses.

The next subsections first introduce the two access kinds, direct and derived, and then the additional features that may appear on top of either.

4.3.1. Access kinds

A stack access is either direct or derived, depending on how its address is obtained.

4.3.1.1. Direct accesses

```
1          ldr     …, [sp, #12]     ; AArch64 load
2          str     …, [sp, #8]      ; AArch64 store
3          mov     …, [rsp + 16]    ; x86-64 load
4          mov     [rbp - 8], …     ; x86-64 store
```

Direct accesses are the simplest case: loads or stores that target a fixed location through an immediate offset from the stack pointer or frame pointer. They are easy to identify, and the accessed stack region can be computed statically. This is usually in the current stack frame, but the same addressing form can also reach into the caller's frame.

4.3.1.2. Derived accesses

```
1          lea     rdi, [rbp - 8]
2          …
3          mov     …, [rdi]         ; access through rdi, resolves to rbp-8
```

A derived access is any load or store that reaches the stack through a general-purpose register rather than through the stack pointer or frame pointer directly. Recognising it as a stack access requires tracing the reaching definitions of the base register and proving that the resulting value resolves to a stack-based address; for instance, when the chain terminates with a `lea` based on SP or FP.

This resolution can involve following more than a single defining instruction: the chain may cross register-to-register moves, arithmetic operations that adjust the address, and memory loads that reload a previously stored stack address. The process may also reach a dead end at an opaque source, such as a function parameter with no available caller information.

Moreover, the resolution often requires crossing basic-block boundaries. In the example above the defining `lea` resides in the same basic block as the access, but more commonly the definition lives in a different block, separated from the access by any number of branches and merges:

```
1  BB0:
2          lea     rsi, [rsp + 8]
3          jne     BB2
4  BB1:
5          …
6  BB2:
7          mov     …, [rsi]        ; access through rsi, defined in BB0
```

Recognising the load in `BB2` as a stack access requires continuing the reaching definition search across blocks back to `BB0`. Most importantly, the resulting resolution is only valid along the control-flow graph paths that actually carry the definition down to the access. Now suppose `BB1` redefines `rsi`:

```
1  BB1:
2          lea     rsi, [rsp + 16]
```

The single load instruction in `BB2` then yields two derived stack accesses, each valid only on the paths that carry the corresponding definition to the load: one at `rsp + 8` on the CFG path `BB0 -> BB2`, and one at `rsp + 16` on the path `BB0 -> BB1 -> BB2`. The analysis must keep track of these constraints so that later reasoning about each recovered access is restricted to the paths where it is valid.

Conditional moves

Conditional moves create the same kind of ambiguity inside a single instruction. If the condition is false, the destination keeps its old value; if it is true, the destination receives the source value. One such example is `cmovz` on x86-64:

```
1          lea     rax, [rsp + 8]
2          lea     rdi, [rsp + 16]
3          cmovz   rdi, rax
4          mov     …, [rdi]        ; access through rdi, either at rsp+16 or rsp+8
```

The final load is a single instruction, but resolving `rdi` yields two derived stack accesses: one at `rsp + 16` when the condition is false, and one at `rsp + 8` when it is true. As in the cross-basic-block case, each access is valid only under the condition that produces it. Alternatives that do not resolve to stack addresses can be ignored.

Moreover, the source of a conditional move may be a memory operand rather than a register; for example, `cmovz rdi, qword ptr [rsi]` also performs a load, which the analysis must reason about separately and treat as another stack access if `rsi` resolves to the stack.

Pointer-to-stack

A further complication in resolving derived accesses is the pointer-to-stack pattern, where the base register is loaded from a stack slot that previously received a stack address:

```
1          lea     rax, [rsp + 8]  ; rax = stack address
2          mov     [rsp + 16], rax ; spill the pointer to the stack
3          …
4          mov     rdi, [rsp + 16] ; reload the pointer
5          mov     …,  [rdi]       ; derived access through the spilled pointer
```

Compared to the basic derived case, recognising this as a stack access requires one extra hop through the stack slot that holds the spilled pointer.

More generally, each additional transformation of a stack-derived value, especially when it escapes into memory, adds another layer of complexity to the analysis.

4.3.2. Additional access features

On top of either access kind, an access may also use vector registers or include indexed addressing.

4.3.2.1. Vector accesses

```
1          movaps  xmm0, xmmword ptr [rsp - 24]        ; direct base
2          movaps  xmmword ptr [rdi], xmm0             ; derived base
```

A vector access is still either direct or derived, depending on its base register, but uses a vector register as source or destination.

4.3.2.2. Indexed accesses

```
1          mov     rax, qword ptr [rsp + 4*rax + 24]   ; direct base
2          mov     qword ptr [rdi + 4*rcx], rax        ; derived base
```

An indexed access is still either direct or derived, depending on its base register, but additionally depends on a second register scaled by a constant.

When the base is the stack pointer or frame pointer, the access is at least known to be stack-relative. If the index can then be proved to hold a fixed value (e.g. set via `mov reg, imm` or `xor reg, reg`), the exact range can be determined, and the access reduces to the direct case. If instead the index is opaque, the analysis can only tell that the access uses the base displacement plus a runtime-dependent offset, but the exact byte range, and sometimes even whether the access stays within the current frame, cannot then be recovered statically.

When the base is another register, it must first be resolved as a derived stack address: if the resolution succeeds, the same index-resolution logic applies; if it fails, no conclusions can be drawn from this access.

The following example illustrates the `rsp`-base case with opaque index:

```c
int f2(int i) {
    int b[4];
    return b[i];
}
```

```asm
; Clang x86-64 -O2 -ftrivial-auto-var-init=zero
f2:
        xorps   xmm0, xmm0
        movaps  xmmword ptr [rsp - 24], xmm0        ; (auto-initialisation of b)
        movsxd  rax, edi
        mov     eax, dword ptr [rsp + 4*rax - 24]   ; load 4 bytes from rsp + 4*i
        - 24
        ret
```

In practice, the index is rarely a constant near the access site, as it often comes from a function argument or a loop induction variable. Most indexed accesses thus fall into the opaque case, though a wider analysis beyond the local function, for example over the program as a whole, may still resolve some.

4.4. Locating variables with the load-oracle

The taxonomy above defines what constitutes a recoverable stack load; the load-oracle then treats those loads as evidence that a stack location belongs to an automatic variable.

The basic idea is that an instruction like `ldr w8, [sp, #12]` represents a clear indication that four bytes of stack memory at a specific location are being used, suggesting they may be part of the memory allocated to a variable. At first glance, this approach to identification may appear both complete and easy to implement, but it merely shifts the original problem to discovering all stack loads that exist in a program. This problem is undecidable in the general case, since whether a load targets the stack depends on the runtime value of its base register; static analysis cannot determine this for all executions and will therefore exhibit false positives, false negatives, or both.

Even when a load is successfully recognised as a stack access, it reveals only the byte range of that particular access, not the full extent of the variable it belongs to: two loads at adjacent offsets could equally be targeting the same variable or two distinct ones. Recovering the total size would require observing every load that touches the variable, which is just as undecidable.

With those limits established, the following subsections examine the remaining load-specific cases that affect the load-oracle.

4.4.1. Inter-procedural loads

The discussion so far has mostly assumed that the load appears in the same stack frame as the variable it refers to; however, a local variable may also be read from a different function,

which requires analysing more than one function at a time. Two inter-procedural cases can be distinguished.

4.4.1.1. Loads through a passed address

When a function takes the address of a local variable (e.g. by pointer or reference) and passes it to another function that uses it, the actual load occurs in the callee:

```
1 void f3() {
2     int c = …;
3     foo(&c);
4 }
```

```
1 f3:
2         lea     rdi, [rsp + …]
3         call    foo
4         …
5 foo:
6         mov     …, [rdi]        ; load accessing f3's stack
```

The function `f3` itself contains no load of `c`, and locating the access therefore requires digging into the callee. The analysis must record that the first argument register `rdi` holds an address into `f3`'s frame on entry to `foo`, then follow how that value flows inside the callee. Here, the passed address may be subject to the same kinds of transformations discussed earlier for derived accesses, or passed on to another function, which further complicates the analysis.

4.4.1.2. Loads into the caller's frame

As a result of calling conventions or optimisations, the callee can sometimes reach directly into the caller's frame:

```
1 foo:
2         sub     rsp, 0x100
3         …
4         mov     …, [rsp + 0x200]     ; accesses beyond foo's own frame
```

Validating such a load requires reasoning in the opposite direction, i.e. from callee to callers. The analysis must enumerate every known caller of `foo`, translate the load's displacement into each caller's frame, and check initialisation there. Any caller it cannot discover (indirect calls, external callers) leaves that code path unvalidated, so the analysis becomes incomplete.

4.4.2. Further limitations of the load-oracle

Beyond the limitations tied to enumerating loads, the load-oracle has additional ones that stem from using loads as an indirect way of inferring variables.

4.4.2.1. Stack slot reuse

As long as their lifetimes do not overlap, compilers are free to place different automatic variables in the same stack slot:

```
1 int f4() {
2     {
3         int d1 = 0;
4         use(d1);
5     }               // d1 goes out of scope, its slot is reused for d2
6     int d2;
7     return d2;      // d2 must be auto-initialised, or this leaks d1
8 }
```

The load-oracle identifies variables purely from their accesses and cannot tell where one lifetime ends and the next begins. As a consequence, the analysis cannot detect if the compiler fails to emit the auto-initialisation of `d2`: since this variable covers the same byte range as `d1`, the earlier store to `d1` is erroneously credited as initialising `d2` too.

4.4.2.2. Padding bytes in widened loads

The compiler may also widen or merge nearby accesses, so a single load ends up spanning multiple variables together with the padding between them. For example, one 8-byte load may cover a 2-byte variable, 2 bytes of padding, and another 4-byte variable, with the two values later extracted through bit operations.

Widening breaks the load-oracle's assumption that every byte in the loaded range belongs to a variable: padding bytes within the widened range are incorrectly attributed to variables. This pattern does not appear to occur frequently in compiled code, so the resulting false positives are likely limited.

4.4.2.3. Non-variable stack loads

Not every stack load is a variable access. Register spills and reloads such as `push rax`/`pop rax` pairs also read the stack, but since the spill always dominates its reload, the slots used by these loads are always initialised and need no special handling.

Other non-variable loads are less benign; examples include a `pop` without a paired preceding `push` emitted for stack-alignment balancing, and explicit Stack Clash probes (e.g. `or qword ptr [rsp], 0` following `sub rsp, 0x1000`). In these cases the loaded value is intentionally ignored, yet the load-oracle still treats them as accesses to verify, producing false positives because the slots are indeed uninitialised.

4.4.2.4. Automatic initialisation eligibility

The load-oracle cannot tell whether a recovered stack slot belongs to a source variable that was meant to be covered by the selected `-ftrivial-auto-var-init` configuration, was explicitly opted out by the programmer, or was excluded by another compiler decision such as a size or count limit. This can produce false positives when validating code generation.

4.4.2.5. Unused variables

A variable that is unused produces no load and is therefore invisible to the load-oracle. In practice, this is rarely an issue: compilers normally optimise them away, and even when they remain in the compiled code, a variable that is never read needs no initialisation.

4.5. Verifying initialisation with the store-witness

Once a stack load has been located, the store-witness determines whether stores prove it initialised.

Under this approach, each byte in the load's range must be written on every relevant control-flow graph path from function entry to the load. A store contributes only if its written byte range can be resolved to the stack. If its base or index cannot be resolved to a stack offset, it cannot be credited, so any load that depends on it appears uninitialised, yielding a false positive.

As the load-oracle requires identifying stack loads in a program, the store-witness relies on identifying stack stores, and the problem is therefore just as undecidable for static analysis, with the same consequences for false positives and negatives.

On top of these constraints, the following subsections examine the remaining store-specific cases that affect the store-witness.

4.5.1. Automatic initialisation code patterns

Smaller variables are typically initialised through ordinary store instructions. Compilers may, however, also implement automatic initialisation through higher-level block patterns. In our tests of GCC and Clang code generation, we observed two such forms, though others may exist:

- calls to `memset`, emitted by Clang for AArch64 and x86-64, and by GCC for AArch64, for larger variables such as arrays above a certain size:

```c
int f5() {
    int e[512];
    return e[0];
}
```

```asm
; Clang x86-64 -O0 -ftrivial-auto-var-init=zero
f5:
        …
        lea     rdi, [rbp - 2048]     ;
        xor     esi, esi              ;
        mov     edx, 2048             ;
        call    memset@PLT            ; (auto-initialisation of e)
        …
```

- uses of `rep stos`, emitted by GCC for x86-64, also for larger variables:

```
; GCC x86-64 -O0 -ftrivial-auto-var-init=zero
f5:
        …
        lea     rax, [rbp-2048]       ;
        mov     rsi, rax              ;
        mov     eax, 0                ;
        mov     edx, 256              ;
        mov     rdi, rsi              ;
        mov     rcx, rdx              ;
        rep stosq                     ; (auto-initialisation of e)
        …
```

  This block-store instruction repeatedly stores the value in `rax` to memory starting at `rdi` for `rcx` times, incrementing `rdi` by 8 bytes on each iteration.

The examples above all use `-ftrivial-auto-var-init=zero`. With `pattern`, the same store shapes may still appear, but the stored value becomes the compiler-chosen pattern rather than zero.

4.5.2. Inter-procedural stores

As with loads, stores may also require inter-procedural reasoning. A compiler may legally omit a variable's initialization whenever it can prove that a callee always performs it on behalf of the caller, possibly through different stores on different CFG paths. To avoid incorrectly flagging such cases as uninitialised, the analysis must cross function boundaries. Consider:

```
__attribute__((noinline))
void f7(int* f) {
    *f = 1;
}
int f6() {
    int f;
    f7(&f);
    return f;
}
```

```
; Clang x86-64 -O2 -ftrivial-auto-var-init=zero
f7:
        mov     dword ptr [rdi], 1          ; initialising store
        ret
f6:
        push    rax
        lea     rdi, [rsp + 4]
```

```
 8          call    f7
 9          mov     eax, dword ptr [rsp + 4]
10          pop     rcx
11          ret
```

Validating the load in `f6` requires following the call to `f7` and finding a store through the passed address. Tracking the argument register in the callee faces the same complications described for loads through a passed address (Section 4.4.1.1). External calls, e.g. into dynamically linked libraries, cannot be followed at all; without supplementary information, the analysis can only treat them as opaque, producing false positives whenever a library function such as `memcpy` actually initialises the slot.

The next chapter turns to how the scanner implements the load-oracle and store-witness in practice.

5. Scanner design and implementation

This chapter presents the implementation of our scanner for validating `-ftrivial-auto-var-init` at the binary level and, more generally, for detecting possible uninitialised stack loads. Building on the taxonomy and concepts introduced in the previous chapter, it opens with a high-level overview, then discusses the main design choices and implementation details, and concludes with a review of its theoretical and practical limitations.

5.1. Overview

The scanner extends LLVM's BOLT-based analyser with a binary-level implementation of the load-oracle (Section 4.2.1) and the store-witness (Section 4.2.2). It relies on BOLT for generic binary-analysis infrastructure, then adds scanner-specific passes for CFG refinement, stack-access recovery, validation, and reporting.

Because it inherits the theoretical limits of both approaches, the scanner cannot be both sound and complete. The implementation therefore favours minimising false negatives, even at the cost of more false positives. When the scanner cannot reach a conclusive result, it emits either a diagnostic report, for a stack load whose initialisation could not be proved, or a limitation report, for an unsupported construct or coverage gap that could not be modelled reliably. Both report kinds are labelled for easier triage.

Conceptually, the scanner proceeds through the following stages:

- it first refines BOLT's reconstructed control-flow graph by propagating information about non-returning calls;
- it then collects stack loads and stores, including derived accesses and block initialisation patterns;
- it next validates loads against stores within each function in parallel;
- it then runs two sequential inter-procedural passes that check whether unresolved loads are initialised by callees or callers; and
- it finally emits diagnostic and limitation reports, after classification and deduplication.

5.2. Building on BOLT

BOLT already provides much of the generic binary-analysis machinery the scanner needs:

- control-flow graph reconstruction, which decomposes binary functions into basic blocks;
- a data-flow analysis[19] that tracks the value of the stack pointer at each program point[20];
- a stack-frame analysis[21] for direct stack loads and stores, recording the instruction and the accessed location;
- a reaching definition analysis, tracking which instructions define or use each register; and

[19] See `StackPointerTracking`.
[20] As an offset relative to the reference DWARF's Canonical Frame Address.
[21] See `FrameAnalysis`.

- call graph reconstruction, dominator and liveness analyses, DWARF-parsing support, and more.

However, the scanner still had to implement several additional mechanisms:

- propagation of information about non-returning calls, to improve the CFG (Section 5.3);
- derived-access resolution, tracing register chains back to the stack or frame pointer to recover non-direct stack loads and stores (Section 5.4.2);
- detection of block initialisation patterns, such as `memset` and `rep stos` (Section 5.4.5); and
- inter-procedural reasoning, to validate unresolved loads across functions (Section 5.6).

Relying on BOLT also leaves two notable limits. Although most of its infrastructure is architecture-agnostic, BOLT's stack-frame analysis does not yet support AArch64, limiting the scanner to x86-64 binaries. The scanner's own code is not inherently architecture-specific, though it is currently more oriented toward x86-64 (e.g. `rep stos`). More importantly, any function for which BOLT cannot reconstruct a control-flow graph or resolve stack-frame data is effectively invisible to the scanner and therefore falls outside the analysis.

5.3. Propagating information about non-returning calls

At present, BOLT's control-flow graph reconstruction does not track which callees are known not to return. A basic block ending in a call to `exit` or `abort` is therefore given a fall-through successor like any other call site. The following excerpt, produced by BOLT for a function with two early-exit branches, shows the issue:

```
.Ltmp5
  Predecessors: .LBB06
    0000000e:  mov     edi, 0x12
    00000013:  call    exit@PLT
  Successors: .Ltmp6

.Ltmp6
  Predecessors: .LFT14, .Ltmp5
    00000018:  mov     edi, 0x34
    0000001d:  …
```

The basic block `.Ltmp5` ends in `call exit@PLT` yet lists `.Ltmp6` as its successor, and `.Ltmp6` correspondingly lists `.Ltmp5` among its predecessors. The edge will never be taken at runtime, but both the scanner and other BOLT analyses will still traverse it, producing control-flow paths that never occur in practice and degrading the precision of later analyses.

To refine the CFG, the scanner runs a fixpoint pass before analysis begins.[22] It starts from a user-supplied list of non-returning symbols and propagates the property through wrapper functions; it also recognises the glibc pattern `error(status, ...)` when the status argument is

[22] See `initNoReturnBBs()`.

statically provable to be nonzero. For each call to a non-returning function, the scanner marks the containing basic block as non-returning and isolates any dead instructions that follow the call into a separate unreachable block. It then removes the block's outgoing edges and removes the block itself from the predecessor lists of its former successors. If every reachable exit of a function ends up non-returning, the function itself is tagged, and another round is run until the CFG stabilises.

5.4. Collecting stack accesses

Before validation begins, the scanner must collect as many loads and stores as possible that can be resolved to stack addresses: the load-oracle relies on stack loads to identify variables, while the store-witness uses stack stores as evidence of initialisation.[23] Following the taxonomy introduced in Chapter 4, this section first covers the two access kinds, direct and derived, and then the higher-level initialisation patterns specific to stores.

5.4.1. Direct accesses

The simplest stack accesses are direct loads and stores that use the stack or frame pointers with an immediate displacement. BOLT already records these accesses, and the scanner uses that information as its reference source of direct accesses.[24] These results also include positive-offset loads into the caller's stack frame, which are validated in the dedicated inter-procedural pass (Section 5.6.2).

BOLT's stack-frame analysis also captures non-variable loads such as alignment-restoring pops and Stack Clash probes. The scanner does not discard them during collection; if they remain unsatisfied, it classifies them heuristically during reporting (Section 5.7.1).

5.4.2. Derived accesses

However, those baseline results do not cover accesses whose memory operand is based on a general-purpose register rather than on SP or FP directly, so the scanner resolves these cases separately.[25] Starting from the access, it traces the base register backward through the set of operations it currently supports: register moves, constant-offset arithmetic (`add`, `sub`, `inc`, `dec`), and `lea` instructions. This search relies on reaching definition analysis, whose precision depends directly on the quality of the underlying liveness analysis.

When the traced register chain spans several basic blocks, the scanner additionally tracks two pieces of information to determine for which CFG paths the recovered access is valid:

- a path constraint, an ordered sequence of basic blocks from the access block to the stack-reaching definition block, following the chosen definition chain; and

[23] Collected accesses are represented by the scanner with `ExtendedFrameIndexEntry` objects, an extension of BOLT's `FrameIndexEntry` that can carry additional metadata such as path constraints and definition chains when needed.

[24] See `collectFIEsWithFA()`.

[25] See `collectFIEsWithDerived()`, `findStackDefForReg()`, and `resolveStackValue()`.

- a set of forbidden basic blocks, namely basic blocks that contain alternative reaching definitions of the tracked register.

Recall the cross-basic-block example from Section 4.3.1.2, where `rsi` was defined in two different basic blocks and used in a third:

```
1 BB0:
2     lea     rsi, [rsp + 8]
3     jne     BB2
4 BB1:
5     lea     rsi, [rsp + 16]
6 BB2:
7     mov     …, [rsi]
```

The load in `BB2` actually represents two distinct stack accesses. The definition from `BB0` (`rsi = rsp + 8`) is live only on the branch that reaches `BB2` directly, so the scanner records the path constraint `[BB2, BB0]` and the forbidden set `{BB1}`. The definition from `BB1` (`rsi = rsp + 16`) yields `[BB2, BB1]`; `BB0` may likewise be recorded as forbidden because it contains an alternative reaching definition, though in this example the CFG cannot re-enter `BB0` after `BB1`. The scanner therefore produces two independent records for the same load instruction, each carrying its own path information for later validation.

One notable current scanner limitation applies to most derived stores with path constraints: validation is only supported when the path is actually unique, i.e. when every basic block of the chain except the stack-reaching definition block has a single predecessor. Non-unique cases generate limitation reports.

5.4.2.1. Conditional moves

Conditional moves fit into the same reaching-definition search used for ordinary derived accesses. When the scanner encounters `cmovcc dest, src` while tracing a derived base register, it explores both possible sources for `dest`: its previous value if the condition is false, and `src` if the condition is true.

```
1 lea     rax, [rsp + 8]
2 lea     rdi, [rsp + 16]
3 cmovz   rdi, rax
4 mov     …, [rdi]
```

This yields two candidate accesses, one through the old value of `rdi` and one through `rax`. The scanner does not model the `cmov` predicate itself; it keeps the candidates separate, each with any path constraint and forbidden-block set accumulated earlier in the chain. As a special case, if `src` is `rsp` or `rbp`, the true path already yields a stack address at the `cmov` itself, so the chain may terminate there.

When `src` is a memory operand, only the false path is supported. The memory operand can still be collected as an ordinary load, but the value it loads is not interpreted as another

stack-derived address. Supporting that case would require integrating conditional moves with the pointer-to-stack logic, so the true path currently produces a limitation report.

The same split is reused when tracing forwarded register arguments in the inter-procedural passes, but with a stricter policy: both branches must resolve to the same displacement from the original argument register, or the forwarded address is treated as ambiguous and the wrapper is not credited as an initialiser.

5.4.2.2. Pointer-to-stack resolution

The scanner partially supports the pointer-to-stack pattern (Section 4.3.1.2), where a stack address is first spilled to the stack and later reloaded into the register that performs the actual access. Recognising the later dereference as a stack access requires following the indirection through the spill slot, so the scanner collects stores before loads and uses them during derived-access resolution.

The resolution is recursive: from the reload, the scanner finds the store that wrote the slot, then resolves the definition of the value stored there. The same pattern may also be chained, with a spilled pointer itself being reloaded and re-spilled:

```
1 lea     rax, [rsp + 8]
2 mov     [rsp + 16], rax     ; hop 1: spill the pointer
3 …
4 mov     rbx, [rsp + 16]
5 mov     [rsp + 24], rbx     ; hop 2: re-spill into another slot
6 …
7 mov     rdi, [rsp + 24]
8 mov     eax, [rdi]          ; derived load reached through two hops
```

Each spill-and-reload round trip counts as one hop, and the recursion depth is capped at a small fixed number to prevent combinatorial explosion.

A separate limit applies once the chain passes through a generic, non-SP/FP register. The resulting dereference can still be recognised as a stack access, but the loaded value is not propagated again as another stack address. Chains of generic loads therefore stop after one such hop.

Finally, the scanner recognises these spills only when the pointer can itself be traced back to SP or FP through the same simple operations as in the basic derived case, such as `lea`, register moves, and constant-offset arithmetic. Stack addresses produced in other forms, for example as direct immediates, are not tracked.

5.4.2.3. Killed spills

One complication in pointer-to-stack resolution is store killing: a later store may overwrite a spill slot before the slot is reloaded, so the scanner must recover the value present at the reload site rather than any earlier value written to the slot. Consider:

```
1 mov qword ptr [rsp + 0x8], rax    ; store1: slot holds the value of rax
```

```
2  mov qword ptr [rsp + 0x8], rbx    ; store2: slot now holds rbx, store1 is killed
3  mov r14, qword ptr [rsp + 0x8]    ; reload: r14 = rbx
4  mov rax, qword ptr [r14]          ; load through r14
```

Resolving what `r14` points to means finding the stores that wrote to the slot. Both stores match, but only the second is still live at the reload; tracing the first yields a candidate pointing into the wrong stack region, and the derived load is then validated against bytes it never accesses. Currently, the scanner recognises the kill when the killing store is in the same basic block as either the killed store or the reload. Kills in an intermediate block are not detected, so the dead store is still traced.

Store killing matters only for pointer-to-stack resolution. Ordinary load validation is unaffected by whether an earlier store is later overwritten, as long as each byte in its range is covered by some store.

5.4.3. Vector accesses

Vector accesses do not require a separate address-resolution mechanism. Once their base is resolved as either direct or derived, the scanner uses the instruction's memory-access size to record the accessed byte range, so vector loads and stores differ only by range width.

5.4.4. Indexed accesses

Indexed stack accesses add a second register, scaled by a constant, on top of a direct or derived base. When the scanner can prove that the index register holds a compile-time constant, for example from `mov reg, imm` or `xor reg, reg`, it folds the scaled value into the displacement and treats the result as an ordinary direct or derived access.[26] When the index remains opaque, the exact range is unknown, so the scanner can only record the limitation.

5.4.5. Recognising initialisation patterns

Some initialisations appear as higher-level patterns such as `memset` calls and `rep stos` block stores. For `memset`, the scanner analyses the destination address and size arguments at each call site.[27] For `rep stos`, it similarly resolves the destination and count parameters, i.e. `rdi` and `rcx`.[28] The covered range of the access is inferred directly from the size argument or, for `rep stos`, from both the count and operand size. As for store collection in general, the scanner only tracks the accessed range but not the value written (see Section 4.2.2).

Two limitations apply to these patterns. First, the size or count parameters must resolve to a single compile-time constant. The scanner accepts multiple reaching definitions only when they all agree on the recovered value; opaque or path-dependent parameters are otherwise rejected, because the scanner does not support validating them together yet. Second, these patterns also inherit the restriction on stores with non-unique definition paths (Section 5.4.2).

[26] See `handleIndexedAccess()`.
[27] See `collectFIEsWithMemsets()`.
[28] See `collectFIEsWithRepStos()`.

5.5. Validating loads within a function

Once all supported stack accesses have been collected, the scanner proceeds to validate loads against stores within each function. This intra-procedural pass parallelises naturally because each function is analysed independently.

5.5.1. Backward CFG traversal

As introduced in Section 4.2.2, under the store-witness approach a load is satisfied only when every byte of its range has been written by at least one preceding store on every CFG path from the function entry to the load. To verify this, the scanner walks the control-flow graph backward from the load and accumulates, along each path, the byte ranges written by the stores it encounters.[29] A path succeeds as soon as the load's range is fully covered. Conversely, the traversal stops at the first path that reaches the function entry without full coverage, since one failing path is enough to fail the load. The failing path is not reported at this stage, but recorded for the subsequent inter-procedural passes, which may still find initialisation for the load in another function.

For derived loads that are valid only on specific CFG paths, the traversal must also respect the path constraint and forbidden basic block set captured at collection time (Section 5.4.2). A path is considered relevant to that load only if, while walking backward from the load, it visits the constrained blocks in the recorded order. Blocks outside the constraint may still appear between them, as long as they are not forbidden. Forbidden blocks are rejected while the traversal is still matching the ordered constraint; once the full constraint has been matched, the traversal moves through any predecessor block freely. If the traversal reaches the function entry before satisfying the whole sequence, that CFG path does not belong to the load being validated.

As described, the algorithm tracks which CFG paths are relevant to a given reaching definition and respects the explicit path constraints, but at the moment it does not reason about branch predicates or general path feasibility. Correlated conditions can therefore leave infeasible failing CFG paths that appear as false positives, even when the variable is actually only used on the branch where it was initialised.

5.5.2. Redundant loads

Once an unconstrained load, i.e. one without a recorded path constraint or forbidden-block set, has been validated, the scanner records the covered range. It can then skip a later unconstrained load when its range is contained within one already validated in a dominating basic block, or earlier in the same basic block, since any stores sufficient for the earlier load are also sufficient for the later one. In principle, this optimisation could also be extended to constrained loads, but the scanner does not do that in its current version.

[29] See `hasSufficientStoresInAllPaths()`, `collectStoresForState()`, `processPredecessors()`, and the worklist state `PathTraversalState`.

5.6. Validating loads across functions

After the intra-procedural pass has completed, two further passes look for initialisation in other functions.

5.6.1. Looking into callees

The first inter-procedural pass implements the case where initialising stores for a load reside in another function (Section 4.5.2). For each unresolved load, it considers direct calls that dominate the load, retains those that pass a stack-derived address relevant to the load range as an argument, and then examines these candidate callees in turn until one is shown to initialise the corresponding region.[30]

When the callee is a function in the binary for which BOLT managed to reconstruct its CFG, the analysis verifies it by walking backward from each exit block toward the entry and looking for stores whose combined range contains the passed region. For the call to count as an initialiser for that region, the region must be covered on every exit path. Inside the callee, the passed address may be transformed further or forwarded to sub-callees, which the analysis follows only up to a controllable maximum depth; by default, this means the direct callee plus one wrapper. Within each analysed function, the scanner reuses the reaching definition mechanism described in Section 5.4.2 to trace stores and forwarded arguments back to the original argument register, but the current implementation applies stricter inter-procedural rules that make coverage slightly narrower.

To determine which callees are passed a stack-derived address, the scanner must also identify how many parameters each callee actually uses. Getting this wrong degrades the analysis: overestimating may allow an unrelated stack address lingering in a register to match a store inside the callee, potentially masking a real uninitialised load; underestimating skips legitimate arguments and can therefore produce spurious reports. To determine how many argument registers to inspect for a given call, the scanner first consults a user-provided signatures file, otherwise parses DWARF debug information when available, and otherwise falls back to the x86-64 System V ABI maximum of six arguments passed in registers. In this pass, no more than those first six register arguments are considered; further arguments, which are passed on the stack, are not analysed here and are handled separately only when they appear as caller-frame loads.

When the candidate callee is external, or is present in the binary but lacks a reconstructed CFG, this verification cannot be performed because the scanner cannot inspect the function body. This is a common source of false positives since functions such as `memcpy` or `read` routinely write to stack buffers passed by pointer. The compiler may treat most such callees as opaque and therefore emit initialisation code, but it may also treat some of them as initialisers and thus omit its own initialisation; for the scanner, however, reconstructing which decision the compiler made may be impossible. To supplement the scanner with information about calls that should be treated as known stores, typically external ones, it accepts a known-stores file describing

[30] See `resolveDeferredCalleeChecks()`, `checkPrecedingCallsForStackInit()`, and `checkCalleeInitializesArg()`.

functions assumed to always store to a given argument. Nevertheless, such annotations must be used with care: treating a call such as `gettimeofday` as a known store could hide a real missed-initialisation bug if the compiler should have emitted the initialisation itself but failed to do so.

5.6.2. Looking into callers

The second inter-procedural pass implements the opposite-direction case, walking outward to callers rather than inward into callees (Section 4.4.1.2). It operates on loads flagged during collection as extending beyond the current function's frame, i.e. positive stack offsets that reach into the caller's frame. Such accesses commonly correspond to stack-passed arguments.

For each such load, the scanner consults the call graph and, for each discovered call site, translates the callee-side offset into the corresponding range in the caller's frame. It then reruns the standard backward CFG traversal for checking initialisation in that immediate caller from the call site toward the caller's entry, but does not continue the search inter-procedurally. The load is considered satisfied only if every analysed caller initialises the mapped range on every path to the call; if instead any caller fails, the load is reported as a caller-frame uninitialised access diagnostic.[31]

This pass is necessarily incomplete. Callers outside the analysed set, including external or undiscovered indirect callers, are skipped; a load may therefore be deemed initialised based only on the subset of callers that were both discovered and analysed. If the call graph yields no caller at all, the load is reported as an unsupported caller-frame access limitation, because there is no concrete caller path to validate.

5.7. Reporting

After the intra-procedural and inter-procedural passes, reporting starts from the issues identified during the analysis. Diagnostic reports cover stack loads that were collected successfully but not proven initialised; limitation reports cover unsupported constructs and accesses that the scanner recognised as relevant but could not model precisely enough to use. The scanner deduplicates both sets so that repeated symptoms of the same issue do not dominate the output.

5.7.1. Diagnostic reports

Diagnostic reports are the scanner's conservative bug candidates. Each one corresponds to a stack read whose address and range were known, but for which the intra-procedural and inter-procedural validation passes failed to prove that every accessed byte was invariably initialised.

Because many diagnostics are expected to be false positives, the scanner heuristically matches them against known patterns and assigns triage categories. These categories never suppress a finding, but help the analyst identify which reports are most likely to be benign:

[31] See `validateCallerFrameLoads()`.

- Alignment pops. A `pop` in an exit basic block where every entry block of the function contains a matching `sub rsp, 8`. This usually reflects stack-alignment restoration rather than a meaningful read.
- Dead-register pops. A `pop` whose destination is provably dead after the instruction, according to liveness analysis. This generalises the previous category.
- Stack Clash probes. An `or qword ptr [rsp], 0` immediately following `sub rsp, 0x1000`. Compilers emit this deliberate read of uninitialised memory as a guard-page probe.
- Possibly initialised in callees. The load could not be validated, but a related stack region appears to be passed by pointer to a preceding call.
- Caller-frame uninitialised. A load accessing the caller's stack frame for which at least one analysed caller does not initialise the accessed range. This category is emitted by the caller-side inter-procedural pass (Section 5.6.2).
- Generic uninitialised. The fallback category.

Example diagnostic report output, slightly edited for brevity:

```
1  Read from possibly uninitialized stack location (possible initialization in
   called function find_entry/1(*2)) in function hash_remove at instruction
   000123bf:  movq  0x18(%rsp), %rax
2      for stack range [-0x18:-0x10)
3      definition chain
4          .LFT1329:
5              000123bf:   movq  0x18(%rsp), %rax
6      uninitialized on path
7          .LBB0214:
8              000123a0:   pushq %rbx
9              000123a1:   movl  $0x1, %ecx
10             000123a6:   movq  %rdi, %rbx
11             000123a9:   subq  $0x20, %rsp
12             000123ad:   leaq  0x18(%rsp), %rdx
13             000123b2:   callq "find_entry/1"
14             000123b7:   movq  %rax, %rdx
15             000123ba:   testq %rax, %rax
16             000123bd:   je  .Ltmp1402
17         .LFT1329:
18             000123bf:   movq  0x18(%rsp), %rax
19             000123c4:   subq  $0x1, 0x20(%rbx)
20             000123c9:   cmpq  $0x0, (%rax)
21             000123cd:   je  .Ltmp1403
```

The report shows a direct load at instruction offset `0x123bf`, together with one control-flow path from the function entry to the load on which no initialising store could be found. The load is classified as possibly initialised in a callee because the scanner notices that the accessed range is passed to the callee `find_entry`, with the `lea` at offset `0x123ad` setting up the argument register before the call.

In this specific instance, `find_entry` was present in the binary, but BOLT could not reconstruct its CFG, so the inter-procedural check could not inspect the callee body. The initialisation therefore could not be verified, and a report was emitted conservatively. Manual analysis later confirmed that `find_entry` always initialised the passed variable, so the report was a false positive.

5.7.2. Limitation reports

Limitation reports address coverage rather than defects. They mark instructions where the scanner identified a construct that could influence the analysis but could not model it precisely enough to use, either due to static analysis limitations or because the case is not yet implemented.

The current types of limitation reports mirror the conservative exits described earlier:

- Non-unique stack stores. A derived store whose address definition reaches the store along a non-unique path (Section 5.4.2).
- Conditional moves from memory. A derived address chain where the true path of a conditional move would require interpreting a memory-loaded value as another stack pointer (Section 5.4.2.1).
- Indexed stack loads and stores. An indexed access whose base resolves to the stack but whose index cannot be reduced to a constant offset (Section 5.4.4).
- Conflicting or unresolvable sizes. A block initialiser such as `memset` or `rep stos` whose size or count cannot be resolved to a single constant (Section 5.4.5).
- Stack arguments exceeding the frame. A caller-frame load for which the call graph yields no concrete caller to examine (Section 5.6.2).

Each limitation points to the function and instruction where the unsupported case was encountered. When the scanner still knows a concrete stack range, the report also includes that range and the definition chain; otherwise the instruction itself is the stable location. This keeps unsupported cases visible without presenting them as possible compiler misses.

Example limitation report output:

```
1 Unsupported cross-BB derived stack store in function version_etc_va at
  instruction     0001a2ae:  movq  %rax, (%rsi)
2     for stack range [-0x60:-0x58)
3     definition chain
4         .LBB0303:
5             0001a27f:   movq  %rsp, %rsi
6         .Ltmp2171:
7             0001a2ae:   movq  %rax, (%rsi)
```

5.7.3. Deduplication

Even after report generation, multiple reports may remain for the same stack location, for example due to separate unsatisfied loads along different CFG paths. The redundant-loads

optimisation only helps cull unconstrained satisfied loads (Section 5.5.2). A final deduplication pass[32] groups diagnostic reports by category and exact accessed range, selects as representative a report whose basic block dominates the current one when available, and otherwise retains the earlier report in traversal order. Limitation reports are deduplicated similarly, by range when one is known and by instruction otherwise.

5.8. Known limitations

This section gathers all known limitations, including both those inherent to the approach and those arising from the scanner's current implementation. As discussed in Section 4.2, the practical relevance of a given limit depends in part on the scanner's use case: a report that is a false positive for strict validation of `-ftrivial-auto-var-init` code generation may still be an interesting finding when the same analysis is repurposed to detect genuine stack uninitialised uses. The section first separates the inherent constraints of the method from the scanner's conservative limits, then highlights those that can hide a real finding.

5.8.1. Inherent constraints of the method

Some limits follow directly from trying to validate `-ftrivial-auto-var-init` with a static binary analysis that does not rely on debug information:

- Recovering every relevant stack load and store is undecidable in a purely static binary analysis (Section 4.4, Section 4.5). The scanner therefore cannot guarantee complete coverage, and some false positives and false negatives are unavoidable, for example on unresolvable derived accesses.
- Register-only variables are not handled. Because the load-oracle starts from stack loads, a variable kept entirely in registers never enters the analysis (Section 4.2.1).
- Stack slot reuse remains a fundamental blind spot (Section 4.4.2.1). When two variable lifetimes occupy the same byte range, an earlier store is incorrectly credited as initialising the later variable.
- Compilers may widen loads so that a single load spans multiple variables together with alignment padding (Section 4.4.2.2). This breaks the load-oracle's assumption that every byte in the loaded range belongs to a variable, and can therefore produce false positives.
- Some stack reads are not meaningful variable uses, such as alignment-restoring pops or Stack Clash probes (Section 4.4.2.3). At the binary level, the load-oracle usually cannot distinguish these compiler-intended artefacts from genuine variable loads, so the scanner relies on heuristics for common cases.
- A binary analysis cannot recover which stack slots were actually meant to be covered by the selected `-ftrivial-auto-var-init` configuration and which were intentionally exempted (Section 4.4.2.4). Such slots still enter the analysis through ordinary stack loads and may therefore appear as diagnostics even when the compiler behaved as configured.

[32] See `deduplicateDiagnostics()` and `deduplicateLimitations()`.

- Unused variables are likewise invisible because they never produce a load, though this should rarely be an issue (Section 4.4.2.5).
- Inter-procedural validation is inherently incomplete whenever the relevant caller or callee cannot be identified or inspected statically, for example because of unresolved indirect calls or external code (Section 4.4.1, Section 4.5.2).
- The store-witness checks only that bytes were written, not what value was written or whether the store came from compiler-inserted initialisation rather than ordinary program logic (Section 4.2.2). It therefore cannot verify that the bytes match the selected `zero` or `pattern` mode, and a miscompiled store that writes the wrong value would still satisfy the check.

These limits set the ceiling of the approach, and cannot be eliminated without changing the underlying evidence the scanner relies on.

5.8.2. Conservative implementation limits

The current implementation of the scanner adds a second layer of limits. Most of them appear as conservative failures to prove initialisation rather than as incorrectly assuming it:

- The scanner currently targets x86-64 binaries only, because BOLT's stack-frame analysis does not yet support AArch64 (Section 5.2).
- Functions for which BOLT cannot reconstruct a control-flow graph or recover stack-frame data cannot be analysed directly by the scanner (Section 5.2).
- Cross-basic-block derived stores are discarded when the path from the register definition to the store site is not unique (Section 5.4.2).
- Pointer-to-stack resolution is capped at a small fixed number of hops, allows at most one generic load through a non-SP/FP register, and does not treat immediate-valued stores as candidate stack addresses (Section 5.4.2.2).
- `memset` and `rep stos` are recognised only when their size or count resolves to a single compile-time constant (Section 5.4.5).
- In a derived register chain, a `cmov` with a memory source operand is not resolved along the path that takes the loaded value (Section 5.4.2.1).
- The backward traversal tracks relevant CFG paths for reaching definitions, but not branch predicates or general path feasibility (Section 5.5.1).
- In inter-procedural argument tracking, a forwarded address passing through `cmov` is followed only when both outcomes resolve to the same displacement from the original argument register (Section 5.4.2.1).
- In the callee-side inter-procedural cases it can inspect, the implementation follows wrappers only up to the configured depth (Section 5.6.1).
- In that same pass, the implementation considers only the six integer argument registers of the x86-64 System V ABI; stack arguments are handled separately only when they appear as caller-frame loads (Section 5.6.1, Section 5.6.2).
- Underestimating the number of callee parameters can result in missed initialisations (Section 5.6.1).
- Caller-side validation checks only immediate callers and does not follow a stack-passed argument across more than one caller frame (Section 5.6.2).

Taken together, most diagnostics should be read first as failures of proof, not as confirmed compiler misses.

5.8.3. Unsound implementation limits

More serious are the few limits that can over-credit initialisation or suppress coverage, thereby hiding a real miss:

- Overestimating which call arguments are live at a call site may let an unrelated stack address still present in a register match a callee store and mask a real uninitialised load (Section 5.6.1).
- In pointer-to-stack resolution, a store killed in an intermediate basic block may still be traced, which can make a derived load appear to target the wrong stack range (Section 5.4.2.3).
- Caller-side validation skips callers outside the analysed set, so a caller-frame load may be discharged based only on the callers that were both discovered and analysed (Section 5.6.2).

The next chapter revisits these limits empirically, examining how many reports the scanner produces and which limits dominate on the evaluation corpus.

6. Evaluation and future work

This chapter reports what the scanner can currently detect on real compiler output and, just as importantly, what it still misses. The evaluation is developer-oriented and focused on a single target, GNU coreutils, rather than aimed at finding vulnerabilities: for this first iteration, the goal was to make common stack-access and inter-procedural initialisation patterns tractable. The remaining reports should therefore be read primarily as a map of the current implementation's behaviour, limits, and priorities for future work.

6.1. Evaluation on coreutils

Throughout the scanner's development, the main evaluation corpus was GNU coreutils.[33] It provides a set of small independent C binaries, with enough variation to be a continuous source of tests and improvements for the scanner without immediately making triage intractable.

6.1.1. Corpus and methodology

The corpus used for benchmarking the scanner implementation was GNU coreutils 9.5. Since this is the first implementation of the approach, the evaluation was designed as a controlled check to see whether the scanner can handle real compiler output and produce useful diagnostics.

This corpus is still deliberately modest. Coreutils mostly consists of small C programs, does not exercise C++ features, and does not capture the full complexity of large software. The results in this chapter should therefore be read as evidence about the scanner's current progress, not as a security assessment of coreutils itself.

The binaries were compiled under several configurations in order to exercise different code-generation patterns. The matrix covers GCC and Clang on x86-64 at `-O1`, `-O2`, `-O3`, and `-O3` with LTO, both with and without `-ftrivial-auto-var-init=zero`. The `pattern` mode was not included because, as discussed in the previous chapters, the scanner ignores the initialisation value. The full scanner invocation and raw per-configuration tables are collected in Appendix B.

The scanner also records several metrics intended to make its own limits visible, such as:

- the number of functions reconstructed by BOLT, and the number that the analysis skipped because BOLT did not provide usable stack-frame information;
- the total number of stack accesses detected, split between direct and derived accesses and between intra-basic-block and cross-basic-block cases;
- development-oriented counters such as the number of non-returning basic blocks and functions, redundant loads, killed stack stores, and CFG paths ignored because they pass through forbidden basic blocks;
- a breakdown of diagnostics and limitation reports by kind.

These counters are used to identify which parts of the scanner dominate the remaining false positives and manual triage effort.

[33] https://www.gnu.org/software/coreutils/

6.1.2. Results and observations

These results should be read together with the scanner boundaries catalogued in Section 5.8. A high report count does not necessarily mean many compiler misses: some reports come from opaque calls, skipped functions, current inter-procedural reasoning limits, and more.

Across the matrix, the scanner processed 1695 binaries without a crash and without any per-function timeout. BOLT reported 302619 total functions, reduced to 193971 (64.10% of the total) after filtering out PLT thunks, which are irrelevant for the analysis. Each coreutils binary is small, so the dynamically linked PLT thunks are numerous compared with the number of functions defined by the binary itself. However, the remaining functions were further reduced to 148834 (49.18% of the original total) after filtering out those without BOLT-provided frame information, which the scanner cannot analyse. The evaluation numbers below consider only this final set of analysable functions.

Coverage also varies by compiler and optimisation mode. In the `-ftrivial-auto-var-init=zero` rows, Clang non-LTO configurations analyse about 60% of functions, while GCC falls from 54.43% at `-O1` to 37.59% at `-O3`. LTO reduces coverage for both compilers: Clang `-O3` with LTO analyses 44.16% of functions, and GCC `-O3` with LTO analyses only 20.98%. Because so few functions reach the analysis on LTO builds, these rows mainly measure how well BOLT recovers functions and frames from LTO binaries, rather than how well the scanner reasons about stack initialisation.

Metric	Without init	With init	Change
Analysed functions	74313	74521	stable
Functions with missing-init reports	486	118	75.8% drop
Deduplicated diagnostics	6292	1070	83.0% drop
Diagnostics per 1000 analysed functions	84.7	14.4	83.0% drop
Raw reports	14016	8038	42.8% drop
Deduplicated limitations	3582	3766	4.9% increase

Table 2: Effect of `-ftrivial-auto-var-init=zero` across the corpus.

The differential result in Table 2 is the clearest signal in the evaluation. Automatic initialisation leaves the number of analysed functions essentially unchanged, but it reduces the aggregate count of functions with missing-initialisation reports from 486 to 118 across the corpus and reduces deduplicated diagnostics from 6292 to 1070. In other words, the flag is visible to the scanner and removes most of the suspicious stack reads from no-init builds.

Raw reports are less clean as a comparison metric than deduplicated diagnostics, because they bundle diagnostics and limitations together before deduplication. A few noisy limitation classes can therefore keep the raw total high even when the missing-initialisation diagnostics drop sharply. In the Clang `-O3` init row, for instance, raw reports go up compared to the no-init row even though diagnostics fall from 694 to 108. Deduplicated diagnostics and the number of functions with missing-initialisation reports are therefore the better measures of whether the scanner is recognising the inserted initialisation stores.

Diagnostic kind	Without init	With init	Interpretation
`generic-missing-init`	1389	255	81.7% drop; local missing-store symptoms mostly disappear.
`possibly-init-in-callee`	4184	125	97.0% drop; auto-init removes most callee-opaque stack-object cases.
`uninit-caller-stack-arg`	578	559	Almost unchanged; this is a caller/callee ownership problem, not a local auto-init problem.

Table 3: Dominant diagnostic classes across the corpus.

The diagnostic breakdown in Table 3 explains what the flag does and does not solve. The largest reduction is in `possibly-init-in-callee`: without auto-init, many stack objects look as if they may only be written by a later call; with auto-init, the local compiler-inserted store usually gives the scanner a direct witness. The `generic-missing-init` class falls as well. By contrast, `uninit-caller-stack-arg` barely moves, because those diagnostics flag reads of stack arguments that live in the caller's frame. Auto-initialisation in the current function says nothing about whether the caller initialised those arguments before passing them in. The remaining diagnostic classes (`alignment-pop`, `stack-clash-probe`, and `dead-register-pop`) are small and almost unchanged under the flag.

The split by compiler shows two distinct effects. First, the compilers lower automatic initialisation differently. In the init rows, Clang accounts for 774 stack `memset` stores and zero stack `rep stos` stores, while GCC accounts for zero stack `memset` stores and 1032 stack `rep stos` stores. This is useful for the evaluation, as the scanner recognises both compilers' preferred implementations and is exercised against both.

Second, GCC leaves a larger residual diagnostic set in the init builds. Clang drops from 233 functions with missing-initialisation reports to 10, while GCC drops from 253 to 108. Deduplicated diagnostic volume falls by a similar order for both compilers, from 3228 to 490 for Clang and from 3064 to 580 for GCC, but after normalising by analysed functions the init GCC rows still produce about 18.2 diagnostics per 1000 analysed functions, compared with 11.5 for Clang. This should mainly be read as GCC-generated code exposing more scanner limitations, or possibly GCC-specific initialisation idioms that the scanner does not yet recognise.

The limitation counters reinforce this interpretation. Across the matrix, the scanner emits 7348 deduplicated limitation reports. Indexed stack accesses dominate this limitation volume, with 3278 indexed-load limitations and 2735 indexed-store limitations, together about 82% of all limitations. Cross-basic-block stack-store limitations are less frequent, at 763, but they are disproportionately important because a small number of unresolved stores can fan out into many remaining reports in optimised code. The init/no-init comparison does not reduce these structural limits: deduplicated limitations remain roughly flat, moving from 3582 to 3766. This is expected, since automatic initialisation adds stores but does not make indexed addressing, ambiguous cross-basic-block store paths, or missing frame information easier to model.

6.2. Limitations and future work

The preceding chapter already catalogued the scanner's boundaries. This section focuses on the limits that matter most for the evaluation, and on the follow-up work that could improve the situation.

6.2.1. Interpreting the current limits

Even a mature version of this scanner would support only limited claims. Static binary analysis cannot recover every stack access perfectly, cannot see variables that never appear as recoverable stack accesses, and can only prove that bytes were written, not that the right value was written. The evaluation is therefore evidence about recoverable stack-resident accesses, not a complete proof that every automatic variable was correctly initialised.

The observed results make that distinction concrete. In no-init builds, the scanner mostly reports loads whose stack object may have been initialised by a callee; in init builds, that class nearly disappears. The remaining diagnostic volume is more tied to caller-frame accesses and to visibility gaps that the flag cannot address. The most valuable follow-up work would therefore be to improve BOLT coverage, so that control-flow graph and frame information are available for as many functions as possible, but this evaluation did not assess how feasible that would be.

The limitation counters point to a different class of work. Since they remain roughly flat after enabling `-ftrivial-auto-var-init=zero`, they mostly reflect stack-access cases the scanner does not yet handle well, such as non-unique cross-basic-block stores, path-dependent block stores, deeper pointer-to-stack chains, conditional moves from memory, and infeasible CFG paths. Improving these cases would mainly reduce false positives.

6.2.2. Next directions

Some follow-up directions would change the project more substantially:

- A DWARF-assisted mode would trade support for stripped binaries for better precision. In the compiler-validation use case, debug information could provide variable extents, register-resident variables, and lifetime information for reused stack slots.
- Differential compilation would compare a baseline binary with one built using `-ftrivial-auto-var-init`. This would identify compiler-added stores more directly, and make the scanner a more targeted code-generation validator.
- Path-feasibility checks would address infeasible CFG paths, another source of false positives seen during scanner testing. These checks could be applied only to candidate failing paths, instead of to every path explored by the scanner.
- Register-resident variable analysis would extend the scanner beyond its current stack-load basis. This would broaden the scope to more automatic variables, but with a different proof problem.
- Return-value tracking would cover callees that return stack-derived addresses. The scanner would need to detect those returned pointers and propagate them in the caller.

- A broader evaluation would test whether the approach remains useful outside the current x86-64/coreutils prototype. Useful targets include AArch64 binaries, C++ programs with exception-handling paths, and compiler regression tests.
- Another evaluation track would run the scanner on binaries built without `-ftrivial-auto-var-init` and check whether it rediscovers known missed-initialisation bugs. This would measure its value as a bug detector, where recall against known cases matters more than diagnostic collapse under the flag.
- Finally, part of the scanner implementation should be reusable: propagation of non-returning calls, derived-register resolution, path-constrained accesses, and other components could support validation of other hardening flags at the binary level, or other BOLT use cases.

6.3. Closing

The evaluation shows that a BOLT-based scanner can collect useful binary-level evidence for `-ftrivial-auto-var-init`, within the stack accesses it identifies. On the coreutils matrix, the scanner runs reliably, recognises both Clang's `memset` and GCC's `rep stos` initialisation patterns, and observes the expected diagnostic drop when automatic initialisation is enabled.

At the same time, the residual reports confirm the precision and coverage limits of this static approach. The next step should be to review results obtained on larger and more complex targets before addressing the known engineering limitations of the current scanner. This review can then guide subsequent work by prioritising the largest remaining sources of noise, whether due to existing implementation constraints or other factors as yet unknown.

Another option to explore is to confine the scanner to analyses assisted by DWARF or differential compilation. This could mitigate some limitations of static analysis, at the cost of restricting the tool to code-generation validation on binaries built under controlled conditions.

A. Full review of compiler flags for security hardening

This appendix presents the results of a broad but cursory review of compiler flags for security hardening, in tabular format, carried out in November 2025. Some of these findings were included and discussed in Chapter 3.

The first table lists all the examined options and outlines the corresponding security mitigations they provide.

Option	Mitigation	Mitigated attacks
`-D_FORTIFY_SOURCE=3`	Fortify glibc	BOF
`-D_FORTIFY_SOURCE=2`	Fortify glibc	BOF
`-D_GLIBCXX_ASSERTIONS`	Fortify libstdc++	OOB access
`-fstack-clash-protection`	Stack Clash protection	Stack-heap collision
`-fstack-protector-strong`	Stack canary	Stack BOF
`-mbranch-protection=bti`	Branch Target Identification (BTI)	CF hijacking
`-mbranch-protection=pac-ret`	Pointer Authentication Code (PAC)	CF hijacking
`-mbranch-protection=gcs`	Guarded Control Stack (GCS)	CF hijacking
`-mbranch-protection=standard`	(shorthand for `-mbranch-protection=bti+pac-ret`)	
`-fcf-protection=branch`	CET Indirect Branch Tracking (IBT)	CF hijacking
`-fcf-protection=return`	CET Shadow Stack (SHSTK)	CF hijacking
`-fcf-protection=full`, `-fcf-protection`	(shorthand for `-fcf-protection=branch+return`)	
`-fzero-call-used-regs=used`	Wipe used registers	Data leak
`-ftrivial-auto-var-init`	Init automatic variables	Data leak
`-fzero-init-padding-bits=all`	Zero init padding bits	Data leak
`-mretpoline`, `-mindirect-branch`	Spectre mitigations	Data leak
`-mspeculative-load-hardening`	Spectre mitigations-Speculative Load Hardening (SLH)	Data leak
`-mfunction-return`	Retbleed mitigations	Data leak
`-mshstk`	CET Shadow Stack (SHSTK)	CF hijacking

Option	Mitigation	Mitigated attacks
`-fno-delete-null-pointer-checks`	Retain null pointer checks	Null pointer dereference
`-fno-strict-aliasing`	Disable strict-aliasing UB	Remove UB
`-fno-strict-overflow`	Disable signed-overflow UB	Remove UB
`-fsanitize=safe-stack`	SafeStack	Stack BOF
`-fsanitize=address`	AddressSanitizer	OOB access and UAF
`-fsanitize=cfi`	Control Flow Integrity (CFI)	CF hijacking
`-fsanitize=shadow-call-stack`	ShadowCallStack	CF hijacking
`-fsanitize=undefined`	UndefinedBehaviorSanitizer (UBSan)	Various
`-fsanitize=object-size`	Object size sanitiser	OOB access
`-fstrict-flex-arrays=3`	Strict flex arrays	OOB access
`-fvtable-verify=std`	Virtual pointer checking	CF hijacking
`-fPIE, -fPIC`	ASLR	Various
`-Wl,--as-needed, -Wl,--no-copy-dt-needed-entries`	Link required libraries only	Weak dependency exploitation
`-Wl,-z,nodlopen`	Restrict dlopen	Weak dependency exploitation
`-Wl,-z,noexecstack`	Data Execution Prevention (DEP)	Code injection
`-Wl,-z,relro,-z,now`	Full RELRO	GOT override
`-Wl,-z,relro`	Partial RELRO	GOT override

Table 4: Mitigation summary of several compiler/linker options.

The second table shows option availability by architecture (x86-64 and AArch64) and compiler (GCC and Clang).

Option	x86-64	AArch64	GCC	Clang
`-D_FORTIFY_SOURCE=3`	X	X	X	X
`-D_FORTIFY_SOURCE=2`	X	X	X	X
`-D_GLIBCXX_ASSERTIONS`	X	X	X	X
`-fstack-clash-protection`	X	X	X	X
`-fstack-protector-strong`	X	X	X	X
`-mbranch-protection=bti`		X	X	X
`-mbranch-protection=pac-ret`		X	X	X

Option	x86-64	AArch64	GCC	Clang
`-mbranch-protection=gcs`		X	X	X
`-fcf-protection=branch`	X		X	X
`-fcf-protection=return`	X		X	X
`-fzero-call-used-regs=used`	X	X	X	X
`-ftrivial-auto-var-init`	X	X	X	X
`-fzero-init-padding-bits=all`	X	X	X	
`-mretpoline`, `-mindirect-branch`	X		X	X
`-mspeculative-load-hardening`	X			X
`-mfunction-return`	X		X	X
`-mshstk`	X		X	X
`-fno-delete-null-pointer-checks`	X	X	X	X
`-fno-strict-aliasing`	X	X	X	X
`-fno-strict-overflow`	X	X	X	X
`-fsanitize=safe-stack`	X	X		X
`-fsanitize=address`	X	X	X	X
`-fsanitize=cfi`	X	X		X
`-fsanitize=shadow-call-stack`		X	X	X
`-fsanitize=undefined`	X	X	X	X
`-fsanitize=object-size`	X	X	X	X
`-fstrict-flex-arrays=3`	X	X	X	X
`-fvtable-verify=std`	X	X	X	
`-fPIE`, `-fPIC`	X	X	X	X
`-Wl,--as-needed`, `-Wl,--no-copy-dt-needed-entries`	X	X	X	X
`-Wl,-z,nodlopen`	X	X	X	X
`-Wl,-z,noexecstack`	X	X	X	X
`-Wl,-z,relro,-z,now`	X	X	X	X
`-Wl,-z,relro`	X	X	X	X

Table 5: Availability of several compiler/linker options.

The third table summarises the estimated flag adoption. This evaluation was discussed at length in Section 3.1, where it was noted that measuring this metric accurately requires a more rigorous study. In addition, for these exploratory results:

- the “Production” column indicates if the flag is intended for production use;
- the “OpenSSF” column indicates if the use of the option is recommended the OpenSSF hardening guide;

- the “Debian”, “Ubuntu”, and “Arch Linux” columns indicate if the flag is used by default to compile packages;
- flags for Debian 13 and Ubuntu 24.04 were retrieved with `dpkg-buildflags`;
- flags for Arch Linux were retrieved by checking `/etc/makepkg.conf`; and
- flags for the “Linux kernel” column were retrieved by compiling version 6.17 after configuring it with `make defconfig`.

Option	Production	OpenSSF	GCC -fhardened	Debian 13	Ubuntu 24.04	Arch Linux	Linux kernel	Adoption (est.)
`-D_FORTIFY_SOURCE=3`	X	X	X		X	X		High
`-D_FORTIFY_SOURCE=2`	X			X				Low
`-D_GLIBCXX_ASSERTIONS`	X	X	X			X		Medium
`-fstack-clash-protection`	X	X	X	X	X	X		High
`-fstack-protector-strong`	X	X	X	X	X	X[1]	X	High
`-mbranch-protection=bti`	X	X		X	X		X[2]	High
`-mbranch-protection=pac-ret`	X	X		X	X		X	High
`-mbranch-protection=gcs`	X							Low
`-fcf-protection=branch`	X	X	X	X	X	X	X	High
`-fcf-protection=return`	X	X	X	X	X	X		High
`-fzero-call-used-regs=used`	X							Low
`-ftrivial-auto-var-init`	X	X	X				X[2]	Medium
`-fzero-init-padding-bits=all`	X	X					X	Medium
`-mretpoline`, `-mindirect-branch`	X						X	Low
`-mspeculative-load-hardening`	X							Low
`-mfunction-return`	X						X	Low
`-mshstk`	X							Low
`-fno-delete-null-pointer-checks`	X	X					X	Medium
`-fno-strict-aliasing`	X	X					X	Medium
`-fno-strict-overflow`	X	X					X	Medium
`-fsanitize=safe-stack`	X							Low
`-fsanitize=address`								N/A

[1] From GCC’s defaults.
[2] Default hardened config.
[3] Reportedly used on Android and for the Chromium browser.

Option	Production	OpenSSF	GCC -fhardened	Debian 13	Ubuntu 24.04	Arch Linux	Linux kernel	Adoption (est.)
`-fsanitize=cfi`	X							Medium[3]
`-fsanitize=shadow-call-stack`	X						X	Low
`-fsanitize=undefined`								N/A
`-fsanitize=object-size`								N/A
`-fstrict-flex-arrays=3`	X	X					X	Medium
`-fvtable-verify=std`	X							Low
`-fPIE, -fPIC`	X	X	X	X[1]	X[1]	X[1]		High
`-Wl,--as-needed, -Wl,--no-copy-dt-needed-entries`	X	X						Low
`-Wl,-z,nodlopen`	X	X						Low
`-Wl,-z,noexecstack`	X	X						High[4]
`-Wl,-z,relro,-z,now`	X	X	X			X		High
`-Wl,-z,relro`	X			X	X			Medium

Table 6: Adoption of several compiler/linker options.

The final table presents a high-level assessment on what the options actually do, along with comments on the perceived feasibility of validating their implementation through static binary analysis.

Option	Overview	SBA feasibility
`-D_FORTIFY_SOURCE=3`	Replace some glibc functions (e.g. `memcpy`, `strcpy`) with variants adding boundary checks on buffer sizes at both compile time and run time.	Check that unsafe functions are used only in a context that can be statically proven to be safe. Dynamic object sizes cannot be evaluated statically, so whether a non-checked call was actually safe cannot always be confirmed: high risk of false positives.
`-D_FORTIFY_SOURCE=2`	Similar to level 3, but only objects whose size is resolvable at compile time are fortified. Accesses to objects with	Simpler than level 3, since only sizes known at compile time are checked. However, accesses to objects whose size

[4]Enabled by default nowadays, unless the linker detects that an input object requires an executable stack.

Option	Overview	SBA feasibility
	dynamically computed sizes (e.g. VLAs, `malloc` results) are not covered.	is only known at run time are left unchecked by the compiler: indistinguishable from missed instrumentation at the binary level.
`-D_GLIBCXX_ASSERTIONS`	Add run-time checks to some libstdc++ functions (for example to abort on invalid array access), which are often inlined.	Detecting the inserted assertions should be easy as each failed check calls an abort helper. Verifying coverage would however require enumerating every bounds-checked libstdc++ access, which after inlining cannot be distinguished from ordinary pointer arithmetic.
`-fstack-clash-protection`	Allocate one page of stack memory at a time, each touched with a probe instruction to prevent jumping over any stack guard page.	Out of project scope (scanner already under development).
`-fstack-protector-strong`	Insert canaries in functions with local arrays, address-taken variables, calls to alloca, and more. Rearrange local variables so that arrays are closer to the canary.	Check that functions matching the `-strong` criteria have a canary check before returning. Requires identifying which functions have arrays, address-taken variables, or calls to alloca at the binary level: difficult without debugging data.
`-mbranch-protection=bti`	Add BTI to function entries and other indirect-branch targets.	Check that every indirect-branch target (function entry or jump-table case) starts with BTI. Enumerating these targets has high false positive risk on code that is never indirectly called in practice.
`-mbranch-protection=pac-ret`	Enable return address signing for non-leaf functions.	Out of project scope (scanner already under development).
`-mbranch-protection=gcs`	Enable Guarded Control Stack, a separate stack to store return addresses.	Out of project scope (hardware feature).

Option	Overview	SBA feasibility
`-fcf-protection=branch`	Enable CET Indirect Branch Tracking (IBT). Add ENDBR64 to function entries and other indirect-branch targets.	Check that every indirect-branch target begins with ENDBR64. Enumerating these targets has high false positive risk on code that is never indirectly called in practice.
`-fcf-protection=return`	Enable CET Shadow Stack (SHSTK). Return addresses are also pushed on a hardware-managed shadow stack. If a return address popped from the regular stack does not match the address stored on the shadow stack, an exception is raised.	Out of project scope (hardware feature).
`-fzero-call-used-regs=used`	Zero call-used registers at function return.	Check that all registers used by the function are zeroed before returning.
`-ftrivial-auto-var-init`	Add pattern or zero initialisation to automatic variables that would otherwise remain uninitialised.	Check that all automatic variables are actually initialised before first use. Requires identifying the variable locations (on the stack and in registers), and checking that each location is written at least once before it is read. Risk of false positives if correct initialisation is difficult to verify, for example when initialisation happens in an opaque callee.
`-fzero-init-padding-bits=all`	Guarantee zero initialisation of padding bits in variable initialisers (structures, unions, arrays of such).	Check that padding bits in structures and unions are zeroed. Difficult or impossible without debugging data to identify padding regions.
`-mretpoline`, `-mindirect-branch`	Replace indirect calls/jumps with a return-based trampoline, which causes any specu-	Check that indirect calls and jumps go through retpoline thunks.

Option	Overview	SBA feasibility
	lative execution to loop over a harmless piece of code.	
`-mspeculative-load-hardening`	Mask values loaded from memory when execution might be on a speculative path, to prevent data leak.	Check that loads through non-constant pointers are properly wrapped. Risk of false positives when the compiler omits hardening because determined to be unnecessary.
`-mfunction-return`	Replace `ret` instructions with a return thunk to mitigate speculative attacks.	Check that functions use return thunks instead of bare `ret` instructions.
`-mshstk`	Enable additional CET Shadow Stack instructions.	This only enables instruction generation.
`-fno-delete-null-pointer-checks`	Prevent the compiler from removing null pointer checks, even if the pointer is dereferenced.	At post-link time, if the check was removed, there is no way to know it was there in the source. Unverifiable through binary analysis.
`-fno-strict-aliasing`	Consider that pointers to different types can alias.	Affects which optimisations the compiler applies. Unverifiable through binary analysis.
`-fno-strict-overflow`	Treat signed integer overflow as wrapping (two's complement), preventing optimisations that assume it does not occur.	Affects which optimisations the compiler applies. Unverifiable through binary analysis.
`-fsanitize=safe-stack`	Split the stack into a safe stack (return addresses, spilled registers, safe local variables) and an unsafe stack (to keep all the rest). Unsafe variables are moved to a separate allocation so overflows cannot corrupt return addresses.	Check that the stacks contain the variables they are supposed to contain, which is difficult or impossible without debugging data.
`-fsanitize=address`	Instrument code to detect a range of memory-related errors at runtime.	Out of project scope (development flag).

Option	Overview	SBA feasibility
`-fsanitize=cfi`	Enforce that indirect calls and virtual calls target functions of the expected type.	Check that indirect call sites are preceded by type-check guards. Requires reconstructing the type hierarchy, which is difficult or impossible without debugging data.
`-fsanitize=shadow-call-stack`	Enable a software shadow stack for return addresses. A dedicated register points to a separate stack where return addresses are saved and verified.	Check that functions save/restore the return address via the shadow stack and that the dedicated register is not misused.
`-fsanitize=undefined`	Instrument code to detect undefined behaviour at runtime.	Out of project scope (development flag).
`-fsanitize=object-size`	Instrument memory accesses to verify they do not exceed the compiler-known bounds of the object.	Out of project scope (development flag).
`-fstrict-flex-arrays=3`	Treat trailing arrays declared as `[0]` or `[1]` as fixed-size, enabling further bounds checks at compile time.	A source-level change that affects how the compiler computes object sizes. Unverifiable through binary analysis.
`-fvtable-verify=std`	Insert run-time checks to verify that vtable pointers have not been corrupted before virtual calls.	Check that virtual-call sites are preceded by VtV runtime guards. Rarely used in practice.
`-fPIE, -fPIC`	Generate position-independent code that allows the binary to be loaded at any address.	Trivially verifiable by examining ELF headers (`ET_DYN`)
`-Wl,--as-needed, -Wl,--no-copy-dt-needed-entries`	Only link libraries for which at least a symbol is used.	Out of project scope (linker flag).
`-Wl,-z,nodlopen`	Prevent the shared object from being loaded via `dlopen` by setting a flag in the ELF.	Out of project scope (linker flag).
`-Wl,-z,noexecstack`	Mark the stack as non-executable via the `PT_GNU_STACK` ELF segment.	Out of project scope (linker flag).

Option	Overview	SBA feasibility
`-Wl,-z,relro,-z,now`	Resolve all dynamic symbols at load time and mark the GOT as read-only.	Out of project scope (linker flag).
`-Wl,-z,relro`	Make certain ELF sections read-only after dynamic linking, but leave `.got.plt` writable.	Out of project scope (linker flag).

Table 7: Assessment of several compiler/linker options.

B. Raw coreutils evaluation results

This appendix collects the raw results obtained by running the scanner on GNU coreutils binaries. These data form the basis of the evaluation and interpretation presented in Chapter 6.

The figures are included to support discussion of the scanner's current state. This appendix does not try to enumerate every environment, toolchain, and build parameter required for complete reproducibility.

The test binaries were built by compiling GNU coreutils 9.5 for x86-64 with both GCC and Clang. The matrix covers `-O1`, `-O2`, `-O3`, and `-O3` with LTO (`lto` rows), each with and without `-ftrivial-auto-var-init=zero` (`init` rows).

The scanner was run with the following flags:

- `-scanners=stackinit`
- `-allow-stripped`
- `-experimental-shrink-wrapping`
- `-assume-abi`
- `-stackinit-timeout=60`
- `-skip-funcs=-start`
- `-noreturn-symbols="exit@PLT,abort@PLT,--cxa-throw@PLT,--assert-fail@PLT,--stack-chk-fail@PLT,gmp-die/1,xalloc-die/1"`
- `-interproc-max-depth=10`
- `-known-stores-file=…`

The known-stores file passed to the scanner provided store summaries for several glibc functions. Using a different file should mainly affect the number of "possibly initialised in callees" diagnostics.

The tables presenting the results progress from corpus coverage to recovered accesses and then to report categories. The first table gives the basic scale of each build configuration and shows how many functions were extracted, how many remained after PLT-thunk filtering, and how many were analysable, given that the scanner relies on BOLT's stack-frame reconstruction.

Config	Total functions	Without PLT thunks	Without no-frame-info
`clang-O1`	20314	13611 (-33.00%)	12125 (-40.31%)
`clang-O1-init`	20314	13611 (-33.00%)	12125 (-40.31%)
`clang-O2`	20278	13599 (-32.94%)	12340 (-39.15%)
`clang-O2-init`	20278	13599 (-32.94%)	12338 (-39.16%)
`clang-O3`	19893	13262 (-33.33%)	12033 (-39.51%)
`clang-O3-init`	20230	13551 (-33.02%)	12296 (-39.22%)
`clang-O3-init-lto`	13222	6741 (-49.02%)	5839 (-55.84%)
`clang-O3-lto`	13219	6738 (-49.03%)	5837 (-55.84%)
`gcc-O1`	21192	14112 (-33.41%)	11535 (-45.57%)

Config	Total functions	Without PLT thunks	Without no-frame-info
gcc-O1-init	21192	14112 (-33.41%)	11534 (-45.57%)
gcc-O2	22451	15370 (-31.54%)	9537 (-57.52%)
gcc-O2-init	22451	15370 (-31.54%)	9525 (-57.57%)
gcc-O3	22720	15638 (-31.17%)	8563 (-62.31%)
gcc-O3-init	22717	15635 (-31.17%)	8540 (-62.41%)
gcc-O3-init-lto	11078	4514 (-59.25%)	2324 (-79.02%)
gcc-O3-lto	11070	4508 (-59.28%)	2343 (-78.83%)

Table 8: Function counts at several stages.

The second table counts the stack accesses and initialisation patterns detected in the analysable functions.

Config	Direct stack loads	Derived stack loads	Direct stack stores	Derived stack stores	memset stores	rep stos stores
clang-O1	29779	317	37901	20	0	0
clang-O1-init	30148	311	41711	20	161	0
clang-O2	30155	283	39472	21	0	0
clang-O2-init	30502	285	43189	22	166	0
clang-O3	31520	328	39142	25	0	0
clang-O3-init	33221	344	44467	62	169	0
clang-O3-init-lto	27635	533	29173	1075	278	0
clang-O3-lto	26213	539	25775	1382	0	0
gcc-O1	32530	427	37082	121	0	2
gcc-O1-init	32525	429	38535	125	0	274
gcc-O2	18544	1322	19022	196	0	3
gcc-O2-init	17291	642	20737	22	0	268
gcc-O3	29750	1871	33050	212	0	4
gcc-O3-init	28368	1188	34199	40	0	278
gcc-O3-init-lto	12765	884	14713	24	0	212
gcc-O3-lto	13275	991	14748	16	0	4

Table 9: Recovered stack accesses and initialisation stores.

The final table shows the total and deduplicated report count, then breaks the deduplicated set down by diagnostic and limitation category. Legend:

- (D1) `generic-missing-init` generic uninitialised loads
- (D2) `alignment-pop` alignment pops
- (D3) `stack-clash-probe` Stack Clash probes
- (D4) `possibly-init-in-callee` possibly initialised in callees
- (D5) `uninit-caller-stack-arg` caller-frame uninitialised loads
- (D6) `dead-register-pop` dead-register pops
- (L1) Stack arguments exceeding the frame
- (L2) Indexed stack loads
- (L3) Indexed stack stores
- (L4) Non-unique stack stores
- (L5) Conflicting or unresolvable sizes
- (L6) Conditional moves from memory

Config	Reports	Reports deduped	D1	D2	D3	D4	D5	D6	L1	L2	L3	L4	L5	L6
`clang-O1`	1276	1035 (-18.89%)	54	0	0	563	110	0	7	191	86	24	0	0
`clang-O1-init`	601	492 (-18.14%)	12	0	0	0	110	0	7	232	106	25	0	0
`clang-O2`	1184	1024 (-13.51%)	41	0	0	558	104	0	15	204	81	21	0	0
`clang-O2-init`	514	420 (-18.29%)	0	0	0	3	104	0	15	197	76	25	0	0
`clang-O3`	1123	943 (-16.03%)	37	0	0	555	102	0	13	154	61	21	0	0
`clang-O3-init`	2546	435 (-82.91%)	0	0	0	4	104	0	13	204	78	32	0	0
`clang-O3-init-lto`	1219	617 (-49.38%)	45	0	0	7	101	0	19	203	165	77	0	0
`clang-O3-lto`	2259	1554 (-31.21%)	113	0	0	889	102	0	19	188	173	70	0	0
`gcc-O1`	1449	1385 (-4.42%)	149	0	25	315	2	0	10	655	222	7	0	0
`gcc-O1-init`	1073	1057 (-1.49%)	118	0	25	11	2	0	10	655	222	14	0	0
`gcc-O2`	2037	1295 (-36.43%)	55	11	21	606	26	8	36	48	451	32	0	1

Config	Reports	Reports deduped	D1	D2	D3	D4	D5	D6	L1	L2	L3	L4	L5	L6
`gcc-02-init`	762	675 (-11.42%)	27	11	19	26	24	8	36	49	437	37	0	1
`gcc-03`	1859	1182 (-36.42%)	62	11	29	630	93	8	144	100	59	45	0	1
`gcc-03-init`	637	536 (-15.86%)	32	11	21	58	85	7	144	97	27	53	0	1
`gcc-03-init-lto`	686	604 (-11.95%)	21	4	20	16	29	5	40	64	256	149	0	0
`gcc-03-lto`	2829	1456 (-48.53%)	878	4	20	68	39	4	40	37	235	131	0	0

Table 10: Breakdown of diagnostics and limitation reports.

Glossary

AArch64 64-bit version of the ARM architecture.

ABI Application binary interface; the low-level contract governing calling conventions and binary compatibility.

ASLR Address Space Layout Randomisation; mitigation that makes memory addresses less predictable at runtime.

AddressSanitizer Runtime sanitiser for detecting memory safety errors such as out-of-bounds accesses and use-after-free.

BB Basic block.

BOF Buffer overflow.

Basic block Straight-line instruction sequence with a single entry and exit.

Block scope Scope created when a block is entered and destroyed when the block is exited.

CF hijacking Attack that diverts control flow to an unintended destination.

CFG Control-flow graph.

CFI Control Flow Integrity; mitigation family that restricts indirect control transfers to valid targets.

Call graph Graph whose nodes are functions and whose edges represent possible calls.

Completeness Property of an analysis that ensures that all relevant cases are identified.

Control-flow graph Graph of basic blocks and possible transfers of control between them.

DWARF Debugging With Attributed Record Formats; standard debug-information format.

Data-flow analysis Static analysis that propagates facts about program state across a control-flow graph.

Dominator analysis Analysis that determines which basic blocks must be traversed before reaching another block.

ELF Executable and Linkable Format; standard binary format on most Unix-like systems.

FP Frame pointer.

False negative Real issue missed by the analysis.

False positive Reported issue that is not actually a problem.

Fixpoint pass Pass repeatedly applied until it stops changing the analysed state.

GOT Global Offset Table; table used by dynamically linked ELF binaries to resolve external symbol addresses.

Infeasible path CFG path that exists syntactically but cannot occur at runtime.

LTO Link-Time Optimisation; inter-procedural optimisation performed during linking.

Liveness analysis Data-flow analysis that determines which registers or variables are live at each point in the program.

OOB Out of bounds.

PIC Position-independent code; code that can execute correctly regardless of load address.

PIE Position-independent executable; executable that can be relocated at load time.

PLT Procedure linkage table; ELF stub mechanism for calling externally linked functions.

Reaching definition analysis Data-flow analysis that determines which earlier definitions may reach a given program point.

Register reload Load that restores a previously spilled value from memory into a register.

Register spill Store of a register value to memory, usually the stack, so the register can be reused.

SBA Static binary analysis.

SP Stack pointer.

Scope Region of a program within which a declared entity is visible and accessible.

Soundness Property of an analysis that ensures it does not accept invalid behaviour as valid.

Stack canary Sentinel value placed on the stack and checked before return to detect some stack-smashing attacks.

System V ABI Calling convention and binary interface used by most Unix-like x86-64 systems.

UAF Use after free.

UB Undefined behaviour; program behaviour for which the language standard imposes no requirements.

Worklist Queue or set of pending states used to drive an iterative analysis.

Wrapper function Function whose main purpose is to forward work to another function with little semantic change.

x86-64 64-bit x86 instruction set architecture.