Vom Quelltext zum ausgeführten Programm

Objektcodeprüfung in sicherheitskritischen C++-Projekten

T.Kaufmann (ITC Embedded UG)

Stand: Februar 2026

Ein Review des Quellcodes untersucht, was ein Entwickler ausgedrückt hat. Der Prozessor führt jedoch den durch Compiler, Linker, Bibliotheken und ABI erzeugten Maschinencode aus. Dieser Artikel erläutert, warum die dadurch entstehende Nachweislücke gerade bei C++ groß werden kann, welche kritischen Pfade eine gezielte Objektcodeprüfung rechtfertigen und wie eine solche Prüfung risikobasiert, reproduzierbar und ohne flächendeckende manuelle Assembleranalyse durchgeführt werden kann.

Die zentrale Frage lautet nicht: „Ist der Quellcode korrekt?“, sondern zusätzlich: „Entspricht das ausgelieferte Programm noch den Annahmen, die wir am Quellcode verifiziert haben?“

1 Die Nachweislücke zwischen Quellcode und Ausführung

In einem sicherheitskritischen Projekt werden Anforderungen, Architektur, Quellcode, Tests und strukturelle Coverage miteinander verknüpft. Diese Kette endet allerdings nicht beim C- oder C++-Text. Zwischen Quelltext und Ausführung liegen mindestens Präprozessor, Compiler, Optimierer, Linker, Laufzeitbibliothek, ABI und die konkrete Zielarchitektur.

Eine Objektcodeprüfung soll deshalb nicht pauschal beweisen, dass der Compiler „richtig programmiert“ ist. Sie untersucht konkrete Eigenschaften, die für einen bestimmten Sicherheitsmechanismus oder kritischen Pfad notwendig sind. Beispiele sind:

- die vorgeschriebene Reihenfolge von Registerzugriffen,
- das tatsächliche Vorhandensein redundanter Prüfungen,
- das Fehlen unerwarteter Funktions- oder Runtime-Aufrufe,
- deterministische Laufzeit und begrenzte Stacknutzung,
- korrekte Speicherbarrieren und atomare Zugriffe,
- die Rückverfolgbarkeit zusätzlich erzeugten Kontrollflusses sowie
- die Übereinstimmung von Test- und Coverage-Annahmen mit dem finalen Programm.

Dabei ist eine wichtige Abgrenzung nötig: Weder die Einstufung eines Systems als ASIL D noch der Einsatz von C++ bedeutet automatisch, dass jede einzelne Maschineninstruktion manuell geprüft werden muss. Die Prüfung ist *risikobasiert*: Ihr Umfang folgt dem kritischen Pfad, der verwendeten Sprachkonstruktion, der Compilerabhängigkeit und der verfügbaren anderweitigen Evidenz.

2 Warum C++ die Lücke vergrößern kann

C++ erzeugt nicht zwangsläufig ineffizienten oder fehlerhaften Objektcode. Im Gegenteil: Gut eingesetzte Abstraktionen können sehr effizienten Code ergeben. Das Problem für den Safety-Nachweis ist die hohe semantische Dichte der Sprache: Eine kleine Quelltextänderung kann umfangreiche, nicht unmittelbar sichtbare Auswirkungen auf Kontrollfluss, Ressourcen und Binärstruktur haben.

2.1 Eine Zeile, viele mögliche Operationen

Die Zuweisung

```
a = b;
```

kann eine einfache Register- oder Speicheroperation sein. Sie kann aber auch einen überladenen Zuweisungsoperator aufrufen, Ressourcen freigeben, Speicher allokieren, Referenzzähler verändern, Elemente kopieren oder verschieben und temporäre Objekte erzeugen. Welche Bedeutung gilt, hängt von Typen, Überladungsauflösung, Templates und Bibliotheksimplementierung ab.

Damit steigt der Reviewaufwand: Die lokale Zeile lässt sich nicht zuverlässig bewerten, ohne den vollständigen Typ- und Lebenszeitkontext zu kennen.

2.2 Versteckter Kontrollfluss durch Objektlebenszeiten

RAII ist ein leistungsfähiges Mittel, Ressourcen zuverlässig freizugeben. Der daraus entstehende Kontrollfluss steht jedoch nicht explizit an jeder Sprungstelle:

```
void process()
{
   Guard lock{mutex};
   Buffer data{128};

   if (!check(data)) {
      return;
   }
}
```

Beim frühen `return` werden `data` und `lock` in definierter Reihenfolge zerstört. Die Destruktoren können ihrerseits Funktionsaufrufe und Seiteneffekte enthalten. Mit Exceptions kommen zusätzliche Abbruchpfade und Unwind-Informationen hinzu. Für Laufzeit-, Stack- und Coverage-Betrachtungen muss dieser implizite Kontrollfluss sichtbar gemacht werden.

2.3 Implizite Konstruktionen und Konvertierungen

Ein scheinbar einfacher Aufruf wie

```
send("ERROR");
```

kann — abhängig von der Signatur — mehrere temporäre Objekte erzeugen:

```
send(Message{String{"ERROR"}});
```

Dadurch können Konstruktoren, Kopien, Moves, Destruktoren und gegebenenfalls Speicherverwaltung hinzukommen. Benutzerdefinierte Konvertierungsoperatoren vergrößern diesen Effekt. Der Code bleibt syntaktisch knapp, während der tatsächliche Ablauf erst durch Typanalyse oder Disassembly vollständig sichtbar wird.

2.4 Templates und erzeugte Instanziierungen

Templates beschreiben Familien von Implementierungen. Zwei Aufrufe können zwei getrennte Funktionen mit unterschiedlichem Objektcode erzeugen:

```
filter.process<ChannelA>(value);
filter.process<ChannelB>(value);
```

Das hat mehrere Konsequenzen:

- Jede relevante Instanziierung muss hinsichtlich ihrer konkreten Typen verstanden werden.
- Coverage einer Instanziierung ist nicht ohne Begründung auf eine andere übertragbar.
- Inlining und Spezialisierung können Funktionsgrenzen vollständig auflösen.
- Eine lokale Änderung an einem Template kann viele entfernte Binärstellen verändern.
- Starkes Metaprogramming erschwert Callgraph, Debugging und Rückverfolgbarkeit.

2.5 Virtuelle Aufrufe und indirekte Sprungziele

Bei

```
controller->shutdown();
```

kann das Ziel erst zur Laufzeit über eine VTable bestimmt werden. Für die Safety-Argumentation sind dann Fragen zu beantworten: Welche konkreten Ziele sind möglich? Sind alle Implementierungen getestet? Ist das Objekt sicher initialisiert? Kann die VTable beschädigt werden? Hat der Compiler den Aufruf devirtualisiert und dadurch andere Annahmen erzeugt?

Indirekte Aufrufe sind nicht grundsätzlich unsicher. Sie erhöhen jedoch den Aufwand für Callgraph, WCET-Betrachtung, Kontrollflussanalyse und Coverage.

2.6 Container und Standardbibliothek

Eine einzelne Containeroperation kann zustandsabhängig sehr verschiedene Pfade nehmen:

```
messages.push_back(msg);
```

Mögliche Folgen sind Kapazitätsprüfung, Allokation, Verschieben bestehender Elemente, Konstruktion, Destruktion und Fehlerbehandlung. Selbst wenn Exceptions deaktiviert wurden, muss geklärt sein, wie Bibliothek und Runtime auf einen Fehler reagieren: Rückgabewert, Handler, Programmabbruch oder undefiniertes Verhalten.

2.7 Zero-Cost-Abstraktion und aggressive Optimierung

Viele C++-Abstraktionen sollen zur Laufzeit keine Mehrkosten verursachen. Das gelingt gerade dadurch, dass der Compiler stark transformiert:

- Funktionen werden inlined,
- temporäre Objekte und Kopien werden eliminiert,
- virtuelle Aufrufe werden devirtualisiert,
- Template-Code wird spezialisiert,
- Berechnungen und Prüfungen werden zusammengefasst,
- unerreichbar erscheinende Zweige werden entfernt.

Der Objektcode kann dadurch ausgezeichnet sein, aber strukturell nur noch wenig dem Quellcode ähneln. Insbesondere scheinbare Redundanz verdient Aufmerksamkeit: Wenn zwei Berechnungen laut Sicherheitskonzept unabhängig sein sollen, kann der Optimierer mathematisch äquivalente Ausdrücke zusammenfassen. Im Quelltext sind dann zwei Prüfungen sichtbar, im Binärprogramm möglicherweise nur noch eine.

2.8 ABI, Startup und Runtime

C++ bringt typischerweise zusätzliche ABI- und Runtime-Mechanismen mit:

- VTables, RTTI und Name Mangling,
- Guard-Variablen für funktionslokale statische Objekte,
- globale Konstruktoren vor `main()`,
- Exception- und Unwind-Tabellen,
- `new`/`delete` und ABI-Hilfsfunktionen,
- Compiler-Runtime für besondere Rechen- und Speicheroperationen.

Ein globales Objekt wie

```
static Driver driver{config};
```

kann automatisch erzeugten Initialisierungscode vor dem eigentlichen Programmeinstieg verursachen. Zu klären sind dann Reihenfolge, Abhängigkeiten zwischen Translation Units, Verfügbarkeit von RAM, Clock und Peripherie sowie das Verhalten bei einem Initialisierungsfehler.

3 C ist nicht automatisch transparent

Auch bei C können Compiler Zugriffe umordnen, Prüfungen infolge von Undefined Behaviour entfernen, Schleifen transformieren oder Runtime-Funktionen etwa für Division und `memcpy` einfügen. Objektcodeprüfung ist daher keineswegs ein reines C++-Thema.

Der Unterschied ist vor allem quantitativ und strukturell:

> **Bei C stammen komplexe Abläufe überwiegend aus explizitem Code. Bei C++ können sie zusätzlich aus Typsystem, Objektlebenszeiten, Templates, Bibliotheken, ABI und impliziten Sprachregeln entstehen.**

Das ist kein Argument, C++ pauschal auszuschließen. Es ist ein Argument für einen bewusst begrenzten Sprachumfang und für zusätzliche Evidenz dort, wo die Abstraktion die Sicht auf den tatsächlichen Ablauf verdeckt.

4 Welche Pfade besonders relevant sind

Eine gezielte Prüfung ist vor allem dort sinnvoll, wo eine kleine Abweichung große Folgen hätte oder wo das Verhalten besonders compilerabhängig ist:

- Startup-, Reset- und Low-Level-Code,
- Interrupt-Service-Routinen und Kontextwechsel,
- Core- und Prozessorkommunikation,
- Memory-mapped I/O sowie Clock-, MPU- und ECC-Konfiguration,
- Watchdog- und Abschaltpfade,
- atomare Operationen, Locking und Speicherbarrieren,
- Diagnosemechanismen und redundante Berechnungen,
- zeitkritische Regel- oder Überwachungspfade,
- hoch optimierte Arithmetik,
- Inline-Assembler und Compiler-Intrinsics,
- Code mit implementation-defined oder möglicherweise undefiniertem Verhalten sowie
- C++-Code mit komplexen Templates, virtueller Dispatch-Logik, versteckten Objektlebenszeiten oder unklarer Bibliothekswirkung.

5 Ein praktikables Prüfverfahren

5.1 1. Prüfgegenstand und Eigenschaft festlegen

Die Aussage „Assembler wurde geprüft“ ist weder reproduzierbar noch aussagekräftig. Am Anfang steht eine konkrete, testbare Erwartung, zum Beispiel:

- Der Ausgang wird spätestens innerhalb von 20 µs deaktiviert.
- Zwei unabhängige Plausibilitätsprüfungen bleiben im Objektcode erhalten.
- Zwischen zwei Registerzugriffen liegt die vorgeschriebene Barriere.
- Der Pfad allokiert keinen dynamischen Speicher.
- Es treten keine unerwarteten indirekten oder Runtime-Aufrufe auf.
- Der maximale Stackbedarf beträgt höchstens 192 Byte.

Erst eine solche Erwartung macht die Objektcodeprüfung zielgerichtet und reviewbar.

5.2 2. Reproduzierbaren Produktions-Build verwenden

Geprüft wird der Build, der ausgeliefert werden soll — einschließlich:

- exakter Compiler- und Linkerversion,
- Optimierungsstufe, CPU-, FPU- und ABI-Einstellungen,
- Präprozessor-Defines,
- Linkerscript und Section-Layout,
- Link-Time Optimization,
- statisch oder dynamisch eingebundener Bibliotheken sowie
- aller sicherheitsrelevanten Buildoptionen.

Ein Debug-Build ohne Optimierung ist kein Ersatz für die Analyse des Produktions-Builds.

5.3 3. Das final gelinkte Programm analysieren

Ein einzelnes Objektmodul kann sich durch LTO, Dead-Code-Elimination, Linker Relaxation oder Bibliotheksauflösung noch verändern. Maßgeblich ist deshalb das final gelinkte ELF beziehungsweise das tatsächlich ausgelieferte Binärprogramm.

Für eine GNU-basierte ARM-Toolchain sind beispielsweise folgende Befehle nützlich:

```
arm-none-eabi-objdump -d -S firmware.elf
arm-none-eabi-objdump -h -t firmware.elf
arm-none-eabi-nm -S --size-sort firmware.elf
arm-none-eabi-readelf -a firmware.elf
```

Ergänzend helfen Linker-Map, Callgraph, Compiler-Optimierungsberichte und Stack-Usage-Ausgaben wie `-fstack-usage`. Die konkrete Eignung und das Vertrauen in diese Werkzeuge sind im Projektkontext zu bewerten.

5.4 4. Quelltext, Anforderung und Objektcode korrelieren

Die Analyse sollte nicht in einem isolierten Assembler-Listing enden. Sinnvoll ist eine dreifache Korrelation:



Für jede notwendige Eigenschaft wird gezeigt, welche Quelltextstelle sie implementiert und wie sie im Objektcode realisiert ist. Nicht zum Quellcode rückverfolgbare Bestandteile werden klassifiziert: Compiler-Runtime, Bibliothekscode, Startup, Alignment, Thunks, Exception-Unterstützung oder tatsächlich unerwarteter Code.

5.5 5. Abweichungen bewerten und automatisieren

Nicht jede Abweichung ist ein Fehler. Inlining, Copy Elision oder Devirtualisierung können erwünscht sein. Entscheidend ist, ob eine Safety-Annahme verletzt oder die Nachweisführung geschwächt wird.

Wiederkehrende Eigenschaften sollten automatisiert werden. Geeignet sind unter anderem:

- Verbotslisten für Heap-, Exception- oder unerwünschte Runtime-Symbole,
- Grenzwerte für Stacknutzung und Funktionsgröße,
- Prüfung kritischer Section-Adressen,
- Erkennung neuer indirekter Aufrufe,
- Callgraph- und Symbol-Differenzen zwischen freigegebenen Builds,
- reproduzierbare Binär- und Mapfile-Berichte.

Eine automatisierte Prüfung ersetzt nicht jede semantische Bewertung. Sie sorgt aber dafür, dass bekannte Eigenschaften bei jeder Änderung erneut kontrolliert werden.

6 Beispiel: sicherheitsrelevante Abschaltung

Angenommen, eine Abschaltung wird durch folgende Bedingungen ausgelöst:

```
if ((temperature > LIMIT) ||
   (plausibility_error != 0u)) {
   disable_output();
}
```

Eine sinnvolle Analyse stellt nicht nur fest, dass Vergleiche und ein Sprung existieren. Sie beantwortet konkrete Fragen:

- Werden beide Bedingungen entsprechend der spezifizierten Semantik berücksichtigt?
- Ist Kurzschlussauswertung zulässig oder fordert das Sicherheitskonzept die Auswertung beider Diagnoseinformationen?
- Ist der Aufruf von `disable_output()` das erwartete Sprungziel oder wurde ein Wrapper beziehungsweise virtueller Dispatch eingefügt?
- Wurde die Funktion inlined und bleibt der Hardwarezugriff dennoch vollständig und korrekt geordnet?
- Enthält der Pfad unerwartete Runtime-Aufrufe?
- Kann der Compiler aufgrund vorheriger Annahmen einen Zweig für unerreichbar halten?

Besondere Vorsicht gilt bei beabsichtigter Redundanz. Zwei im Quellcode getrennt formulierte, mathematisch äquivalente Berechnungen sind nicht automatisch zwei unabhängige Berechnungen im Objektcode. Soll die Redundanz einen zufälligen Hardwarefehler erkennen, müssen Architektur, Datenpfade und Compilerverhalten gemeinsam betrachtet werden.

7 Verhältnis zu struktureller Coverage

Source-Level-Coverage kann für den Nachweis ausreichend sein, sofern sie die tatsächliche Programmstruktur angemessen repräsentiert und zusätzlicher Objektcode nachvollziehbar bewertet wurde. Erzeugt der Compiler jedoch Kontrollfluss, der im Quelltext nicht sichtbar ist, entstehen Fragen:

- Woher stammt der zusätzliche Pfad?
- Ist er erreichbar und sicherheitsrelevant?
- Wird er durch vorhandene Tests ausgeführt?
- Ist er Bestandteil einer bewerteten Runtime oder Bibliothek?
- Kann er zu einem Verhalten führen, das die Quellcode-Coverage nicht abbildet?

Das bedeutet nicht, dass automatisch vollständige MC/DC auf jeder Maschineninstruktion verlangt wird. Es bedeutet, dass die Abwesenheit einer Source-Zeile keine ausreichende Begründung ist, zusätzlichen ausführbaren Code zu ignorieren.

8 Dokumentation und Änderungswirkung

Ein kompakter Objektcode-Prüfbericht sollte mindestens enthalten:

Prüfpunkt	Erwarteter Inhalt
Identifikation	Softwareversion, Commit, Build-ID und Prüfsumme
Toolchain	Compiler, Linker, Bibliotheken und relevante Optionen
Prüfgegenstand	Funktion, Pfad und zugeordnete Safety-Anforderung
Prüfziel	Konkrete erwartete Binär-, Zeit-, Stack- oder Zugriffeigenschaft
Evidenz	Listing, Mapfile, Callgraph, Messung oder automatisierter Bericht
Abweichungen	Beschreibung, Auswirkung und technische Bewertung
Ergebnis	Reviewer, Datum, Freigabe oder offene Maßnahme

Die Evidenz ist buildabhängig. Änderungen an Compiler, Optimierungsgrad, Bibliotheken, Templates, Linkerscript, LTO oder relevanten Quelltextstellen können die Prüfung ungültig machen. Im Änderungsprozess muss daher festgelegt sein, wann eine erneute Analyse erforderlich ist.

9 Kompakte Review-Checkliste

1. Ist der kritische Pfad eindeutig identifiziert und einer Anforderung zugeordnet?
2. Ist die zu prüfende Eigenschaft konkret und messbar formuliert?
3. Entspricht der Build exakt der produktiven Konfiguration?
4. Wurde das final gelinkte Programm statt nur eines `.o`-Files betrachtet?
5. Sind implizite Konstruktoren, Destruktoren, Konvertierungen und temporäre Objekte bekannt?
6. Sind alle möglichen virtuellen Sprungziele und Template-Instanziierungen berücksichtigt?
7. Gibt es Heap-, Exception-, Runtime- oder Bibliotheksaufrufe im Pfad?
8. Bleiben beabsichtigte Redundanz, Zugriffsreihenfolge und Barrieren tatsächlich erhalten?
9. Stimmen Stack-, Laufzeit- und Callgraph-Annahmen mit dem Objektcode überein?
10. Ist zusätzlicher, nicht direkt rückverfolgbarer Objektcode bewertet?
11. Decken Tests und Coverage den tatsächlich möglichen Kontrollfluss ab?
12. Ist festgelegt, welche Änderungen eine Wiederholungsprüfung auslösen?

10 Fazit

C++ kann auch in hochkritischen Systemen beherrschbar eingesetzt werden. Seine Stärke — ausdrucksstarke Abstraktion — ist zugleich eine Herausforderung für die Safety-Nachweiskette. Templates, Objektlebenszeiten, virtuelle Aufrufe, Standardbibliothek, ABI und aggressive Optimierung können den Abstand zwischen sichtbarem Quelltext und ausgeführtem Programm erheblich vergrößern.

Die angemessene Antwort ist weder blindes Vertrauen in den Compiler noch eine vollständige manuelle Prüfung des gesamten Binärprogramms. Zielführend ist eine risikobasierte Objektcodeprüfung mit klaren Prüfzielen, reproduzierbarem Produktions-Build, Analyse des final gelinkten Programms und automatisierten Regressionschecks.

> **Quellcode-Review zeigt, was der Entwickler ausgedrückt hat. Objektcodeprüfung zeigt an kritischen Stellen, was Compiler und Linker daraus gemacht haben. Erst beides zusammen schließt die relevante Nachweislücke.**

Hinweis: Dieser Artikel beschreibt eine technische Vorgehensweise. Der konkrete Nachweisumfang ist aus Sicherheitskonzept, Projektprozess, Toolvertrauen, Kritikalität und anwendbaren Normanforderungen abzuleiten.