embeddedworld
Exhibition&Conference

Automatic Verification of (un)intended Data and Control Flows in Embedded Software

Martin Becker
The MathWorks GmbH
Munich, Germany
mbecker@mathworks.com

Jacob Palczynski
The MathWorks GmbH
Aachen, Germany
jpalczyn@mathworks.com

***Abstract*—Data and control flows are essential to software architecture and are commonly defined during the design phase. Yet, they are rarely verified later during implementation. Instead, most verification activities focus on guideline compliance, functional tests, and robustness. In this paper, we dive deeper into the topic of flow analysis and discuss which methods can be used to reliably detect unintended behavior and prevent functional errors, such as forbidden communication between processes of different criticality, accidental disclosure of sensitive data, and even algorithmic errors. Using concrete examples, we explain the different types of flow dependencies and introduce the method of specification-driven code slicing, which can automatically verify the absence of certain unintended flows. This addresses certification requirements from automotive engineering ("Freedom from Interference"), aerospace ("Data and Control Coupling"), and industrial automation ("Restricted Data Flow") by asserting the safety and security properties of software applications. The outlined method can significantly reduce the number of issues discovered late in testing campaigns.**

Keywords—data flow, control flow, interference, static analysis, formal verification.

I. Introduction

Software plays a fundamental role in numerous devices, from consumer electronics, industrial controls, and medical devices to rail, air, and space technology. Many systems have reached a complexity where programming errors are not only possible but likely: Statistics suggest that developers typically make 10 to 50 errors per 1000 lines of code [1]. Many of those errors are caught later by checking compliance to coding guidelines and running unit tests. However, there is one class of defects that is rarely tested and yet can severely impact safety and security properties, namely errors in flow dependencies. This paper focuses on this less-examined error class and explain which methods and tools can be used to eliminate certain flow errors reliably.

Fundamentally, there are two types of flow dependencies, typically occurring in combination: A *control (flow) dependency* (see Figure 1a) exists when the execution of an instruction (here `y:=12`) depends on the value of an expression (here `x`), and a *data (flow) dependency* (see Figure 1b) exists when the value of an expression (`z`) depends on the value of another (`a` and `b`, as well as indirectly on `read_input`).

```
1 |   if (x > 10) then
2 |       y := 12
3 |   end if
```

(a) Control flow dependency

```
1 |  a := read_input()
2 |  b := 5
3 |  z := (a * b) / 2
```

(b) Data flow dependency

Figure 1: Illustration of the two types of flow dependencies.

The sum of all flow dependencies drives the logical behavior of a software. Therefore, flow errors can lead to undesired behavior and, as we will demonstrate later – even to crashes. It should be noted that such errors cannot be verified by checking coding guidelines like MISRA C™ or through robustness testing. The intended behavior is not evident from the source code but rather captured in high-level specifications such as software design documents. Moreover, even robust, and functionally correct software can still suffer from flow problems, for example by exposing sensitive data and creating a security vulnerability (e.g., Heartbleed [2]).

Verifying proper flow dependencies is essential when processes with different levels of criticality shall be segregated. For example, in medical devices, it is vital to isolate the logging of patient data from the operation of an insulin pump. In a road vehicle, the separation between the audio system and the driver assistance systems is essential. In both cases, unintended interferences could lead to hazardous outcomes. Therefore, industrial standards typically require an analysis of data and control flows [3] [4] [5] to obtain evidence of correct implementation. Unfortunately, they provide limited detail regarding which aspects of data and control flows shall be verified. This paper aims to draw a clearer picture of the demands and utility of flow analysis.

A. Challenges

Typical development tools such as compilers, debuggers, and unit tests have limited detection capabilities regarding the issues mentioned earlier. They can only identify the shallower flow errors and can merely confirm their *existence* rather than prove

their *absence*. To thoroughly identify errors in control and data flows, we need tools that operate directly on the code base—where the semantics remain traceable—and consider the design specifications. Such advanced tools are essential for a more accurate and comprehensive error detection process.

Development environments (e.g., IDE) come to mind for this purpose. However, they are inadequate, too. Functionality such as reference search (e.g., "Find all References" in Microsoft VS Code™) only shows direct access to variables. It may miss transitive effects like propagation into function parameters and indirect access via pointers ("Alias"). Other information, such as the call hierarchy, is not necessarily complete, either, especially in the presence of complex constructs like polymorphism and function pointers. As a result, data and control flows cannot be fully tracked and verified in the development environment.

```
1| int main(int argc, char** argv) {
2|   const int A = 42;
3|   char buf[3];
4|   printf("A=%d, argc=%d\n", A, argc);
5|   buf[argc] = 66;
6|   printf("A=%d\n", A);
7|   return 0;
8| }
```

```
martin@mathworks$ gcc ubtest.c
martin@mathworks$ ./a.out 2 3
A=42, argc=3
A=66
martin@mathworks$
```

Figure 2: Undefined behavior may create invisible data and control flows.

Another pitfall is *Undefined Behavior* (UB)–code constructs that, by definition of the respective programming language (e.g., the C99 standard [6]), can have arbitrary side effects. These can create invisible dependencies beyond the semantics of the programming language and may cause significant testing difficulties. The example in Figure 2 illustrates such UB. Although variable `A` is declared as a constant with a value of 42, it unexpectedly carries the value 66 at the end of the program. The reason is the assignment on line 5, which represents UB. As `argc` is three and thus outside the bounds of `buf`, a buffer overflow happens. The value assigned to `buf[argc]` is actually written to the memory area of `A`, as it is directly adjacent to `buf`. Hence, an invisible dependency exists between the variables `argc` and `A`. To avoid such semantic gaps, UB must be removed entirely.

Static code analysis tools can provide deeper insights into the code and even identify UB. Moreover, they can identify errors in control and data flows like dead branches or writing to local variables without further use. Such tools are highly recommended to fix defects early and improve the overall code quality. However, they cannot verify flow correctness on their own but merely support a manual flow analysis. In the following, we list the requirements for reliable and precise verification of control and data flows to ensure all these challenges can be addressed.

B. Requirements for Reliable Flow Analysis

A method for the reliable analysis of data and control flows must fulfil the following requirements:

- **Identify undefined behavior ("no semantic gaps")**: To avoid invisible data and control flows, UB must be identified so the developer can remove it.
- **Soundness ("no missed flows")**: Every feasible program state must be modeled. This requires, among other things, pointer analysis (Alias Analysis) to capture indirect accesses and a semantically complete model of the programming language, including its standard library.
- **Precision ("no needless warnings")**: Variable values and their relations must be calculated as precisely as possible. Towards this, a value analysis is necessary, which models all operations and expressions, including arithmetic, as precisely as possible.
- **Tracking causal chains**: Flows should be traceable across function calls (Interprocedural Analysis), separable in overlapping execution paths (Context Sensitivity), and complete in the presence of multitasking.
- **Qualified**: In the context of industrial standards (e.g., ISO 26262 [3] or IEC 61508), the tool should be validated and qualified, so that its outputs can be used to claim certification credits.

Tools that meet these requirements are typically called *Slicers based on Formal Methods* and exist for both model-based development (e.g., [7]) and for source code (e.g., [8]).

II. Analysis Approach

In this section we explain the technology of (sound) Program Slicing, which satisfies the methodological requirements discussed in the previous section. For brevity, we limit the explanation to code level, but the technology has been extended towards higher-level design models (e.g., Simulink [7]), too.

To conduct a precise, sound, and transitive flow analysis, the following two steps are required:

A. **Perform Sound Semantic Analysis:** Compute all possible variable values, flow decisions, and pointer targets that are feasible in the program, including instances of undefined behavior.

B. **Construction of a Program Dependency Graph:** Leverage the data from the previous step to capture transitive flows between all program locations and enable end-to-end flow discovery and verification.

We provide more details on each of the two in the following:

A. Perform Sound Semantic Analysis

The first step towards flow verification is to analyze the source code for possible control and data flows. This includes the analysis of variable values (since they drive control flow decisions) and interprocedural analysis (callers, parameter values, effects of callees, and return values). While many software analysis approaches exist, only a few satisfy the soundness and precision requirements stated in the previous section.

Abstract Interpretation [9], a member of the family of formal verification methods, is a suitable analysis technique for large-scale software applications. When implemented consistently and used correctly, it offers soundness by exhaustively considering all possible program states [9, p. 242]. In instances of

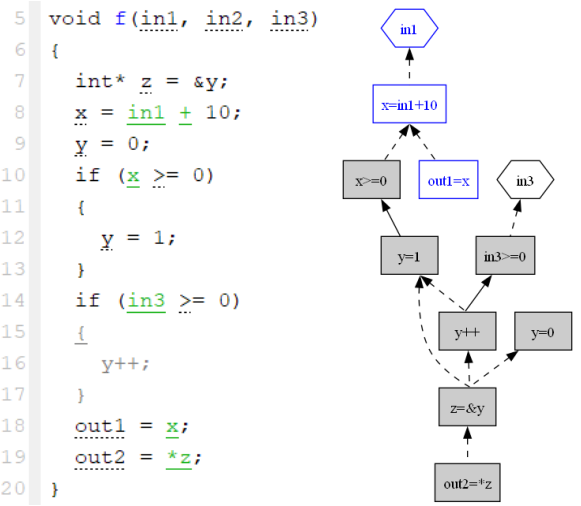

Figure 3: Source code and its *Program Dependency Graph*.

uncertainty, such as when contextual information is lacking or operands are unknown, Abstract Interpretation overapproximates, i.e., assumes that the possibility in question is feasible. To achieve soundness, it must also resolve all pointers ("Alias Analysis"), as they can alter variable values and control flows indirectly.

As a beneficial side effect, sound Abstract Interpretation can identify when parts of the program are logically unreachable. Figure 3 shows that if the value analysis deduces that `in3` is strictly negative, then the increment `y++` on line 16 becomes unreachable. Thus, there is no data dependency on line 16 and no control dependency on line 14. This information will be leveraged later.

Finally, sound Abstract Interpretation can identify undefined behavior. As discussed before, this is a prerequisite to trust the semantics of the programming language. More details on that, as well as its effects, are given in [10].

B. Construct Program Dependency Graph

Once the sound semantic analysis is complete, the deduced information is used to transitively chart all data and control flows through the entire software. We leverage a method called *Program Slicing*, as proposed by Mark Weiser [11]. It builds a program dependency graph (PDG) that captures all feasible control and data flows in one view, as shown in Figure 3: Nodes represent variable reads or writes, and edges represent direct dependencies on preceding nodes. Moreover, there are two types of edges: a) data flow edges, depicted with dashed lines, and b) control flow edges, depicted with solid lines. For example, the PDG in Figure 3 shows that `out1` on line 18 has a data dependency on the assignment of `x` on line 8, which in turn has a data dependency on `in1`. Similarly, we can see that `out2` has a data dependency on the assignment of `z`, and that `z` itself has multiple data dependencies, which are moreover control-dependent on the values of `x` and `in3`. Constructing such a graph is straightforward once semantic analysis has been completed.

Before we use the PDG for verification means, we can further simplify it thanks to sound and precise information from the semantic analysis. Since we know that line 16 in Figure 3 is unreachable (see previous section), we can safely remove the associated node in the PDG. This immediately cuts off the control dependency `in3>=0` and the data dependency to `in3`. Similarly, more edges can be removed if we have precise information about pointers. Effectively, this step reduces needless warnings.

We now have a complete graph that precisely captures data and control flow dependencies. Verifying a specific dependency becomes a reachability problem, which can be solved with standard graph algorithms. This graph can be used in forward direction (show downstream impact) or backward direction (show influencing parts). To distinguish between data or control flows, the according edge type can be removed prior to reachability analysis.

In the next section, we explain how this analysis approach can be used in development workflows.

III. Workflow in Practice

The technology can be used in two fundamentally different ways, namely:

A. Interactive exploration, and

B. specification-driven flow verification.

Both approaches differ in application areas and workflows.

A. Interactive Exploration

Most users starting with data- and control flow analysis have no specific expectations regarding the results. They focus on discovering existing flows and understanding the program logic. One obvious scenario is to better understand legacy software.

Typically, explorative use starts by selecting start or end points. The computed flows from/to these points are highlighted in the design artifact. Examples are shown in Figure 4 for both

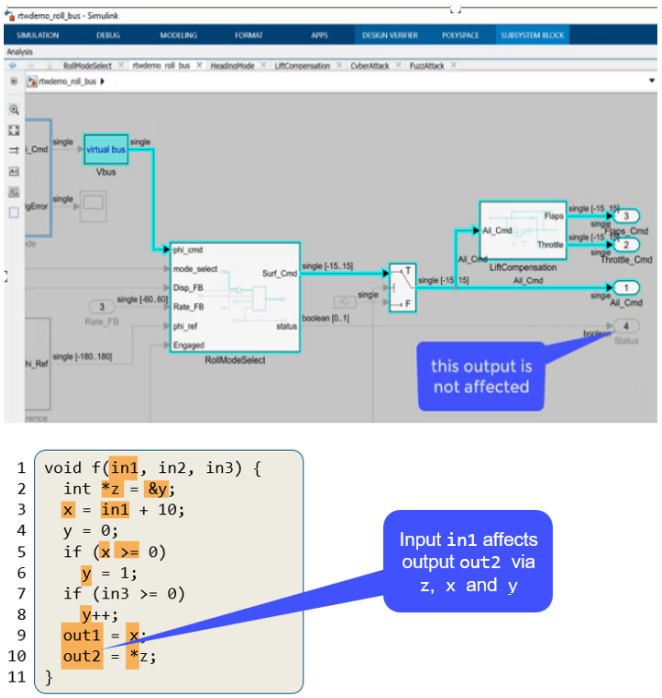

Figure 4: Interactive, explorative dependency analysis on models and code.

model and code. The user can interactively walk through the software and understand the logical coupling between various program locations and variables.

Exploration can also go beyond discovery. It can be applied for impact analysis (downstream effects) or taint analysis (upstream influences). This can be leveraged to assess the criticality of a certain variable, component, or signal in the context of the whole software. Assume, for example, that the flow analysis shows that a variable can have downstream effects on the braking system of a vehicle. The user can now deduce that this variable is safety-critical and consider implementing additional protective code. Similarly, if a security problem appears in one part of the software, explorative use can help identifying the potential damage that follows.

These are just some of many possible use cases for interactive exploration. From now on, we will focus more on the specification-driven analysis.

B. *Specification-driven Verification*

The interactive exploration reliably captures all flows, yet it is still "blind" to the specification. Thus, it cannot identify unintended behavior and wrong implementations. We address this limitation by adding the notion of *(un)expected flows* as an analysis input. The analysis can automatically prove or disprove these, thereby identify erroneous flows in the code. This relieves developers from the burden of manually exploring the model or code. Instead, they only will focus on deviations, which will be explained to them with specific counterexamples. If flows are proven to be compliant with the expectation, no further steps are required, and certification credits can be claimed.

The specification-driven analysis generally consists of the following three steps:

1. **Define (un)expected flows:** First, declare mandatory or forbidden flows in a machine-readable format (e.g., XML file). Typically, this is done pairwise (A must not influence B) or group-wise (e.g., the set of critical variables must have no data dependency on the set of uncritical variables).
2. **Run flow analysis:** The analysis is executed as described in Section II. The specifications from the previous step are converted into reachability objectives and evaluated on the dependency graph. For compliant objectives, no further action is necessary. For each unexpected yet feasible flow, the analysis generates a counterexample: one complete trace exposing the unexpected flow. Each flow that is expected yet absent is flagged for further review.
3. **Review of deviations:** Since the analysis is based on formal methods (see Section II), only deviations from expectation need review. In case of an unintended flow, the counterexample (see "read of username," orange in Figure 1) makes debugging easy and efficient. For missing flows (see "write to calibrated," red), counterexamples are not available for obvious reasons. The reviewer will have to analyze the cause of the missing flow.

The specification step can be automated for some well-known patterns (e.g., LDAP injection, missing input validation), requiring no additional effort for standard flows. However, the real power of this specification-driven usage comes with custom patterns that are typically known from the design but never checked. One example is components with different safety levels (e.g., ASIL in ISO 26262 [3]). Components with higher criticality must not depend on lower-criticality components ("Freedom from Interference"). This objective allows to automatically generate specifications for unexpected flows.

This specification-driven verification is well-suited for CI pipelines, since it generates a list of deviations to be reviewed alongside the findings of ordinary guideline checkers.

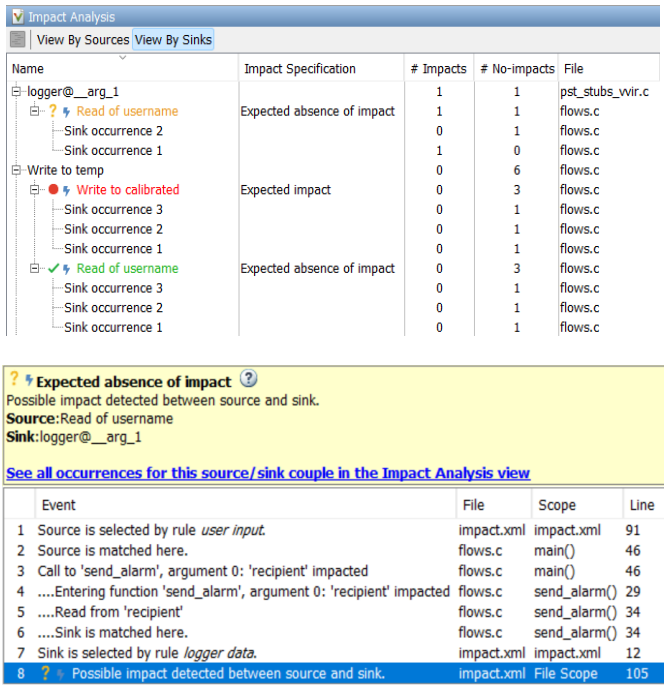

Figure 5: Results of a specification-driven flow analysis and one deviation.

IV. EXAMPLES

To illustrate the specification-driven analysis approach, we use a temperature monitoring program, shown in Figure 6. It reads a sensor, calibrates the measurement value, and sends an email notification if the temperature becomes too high. We want to verify the following objectives for the control and data flows:

1. Measurement values are unaffected by user inputs.
2. The measurement result is always calibrated.
3. Confirm the absence of exploitable security vulnerabilities in the program.

We will analyze these objectives using *Polyspace Code Prover* [8], a sound abstract interpretation tool. Polyspace Code Prover enables users to prove the absence of undefined behavior and verify the control and data flows in C/C++ code.

```
 9| #define PERIOD 1
10| #define CRITICAL 30
11|
12| float calibrate (float rawval, unsigned t) {
13|     const int calibrated = rawval * 0.27392f + 273.15;
14|     logger("sens is %.1f", calibrated);
15|     return (t > 0) ? calibrated : 0;
16| }
17|
18| int fast_sensor_read (unsigned* const t, int mode_pp) {
19|     (*t) += PERIOD;
20|     int temp = raw_sensor();
21|     if (mode_pp) {
22|         // temp =
23|         calibrate(temp, *t);
24|     }
25|     return temp;
26| }
27|
28| static char cmdbuf[200]; // not on stack
29| void send_alarm (const char*const recipient, int temp, unsigned t) {
30|     if (0 < sprintf(cmdbuf,
31|                     "echo '%d°C at t=%u' | mail -s 'Overheating' %s",
32|                     temp, t, recipient))
33|         (void) system(cmdbuf);
34|     logger("Sent email to %s", recipient);
35| }
36|
37| int main (int argc, char** argv) {
38|     const char* const username = argc > 1 ? argv[1] : "mbecker@mathworks.com"
39|     unsigned time = 0; int temp = 0;
40|     int do_pp = get_config("raw_mode");
41|     do {
42|         sleep(PERIOD);
43|         temp = fast_sensor_read(&time, do_pp);
44|     } while (temp < CRITICAL);
45|
46|     send_alarm(username, temp, time);
47|     return time;
48| }
```

Figure 6: Example program code.

A. *Absence of Undefined Behavior*

As stated earlier, the absence of undefined behavior is a prerequisite for control and data flow analysis. A first verification run exposes undefined behavior on line 32. The e-mail address provided by the user could be too long, causing a buffer overflow and crashing the program. It could even inject malicious code into our program and change the control flow.

We can fix this potential undefined behavior by replacing `sprintf` with `snprintf`. Re-running the verification on the fixed code confirms the absence of undefined behavior (Figure 7). We can now rely on the programming language's semantics.

B. *Proof of Flow (in)dependency*

Our first verification objective ("user input must not affect measurement data") is particularly interesting, since it is an example of interference analysis from various safety standards ("Freedom from Interference" from ISO26262 [3], "Data and Control Coupling" from DO-178C [4], and "Restricted Data Flow" from IEC 62443 [5]). The first step towards verification is specifying an expected data dependency as "(source) `username` must not impact (sink) `temp`." Subsequently, the tool can automatically analyze the data flows and identify deviations. For our example

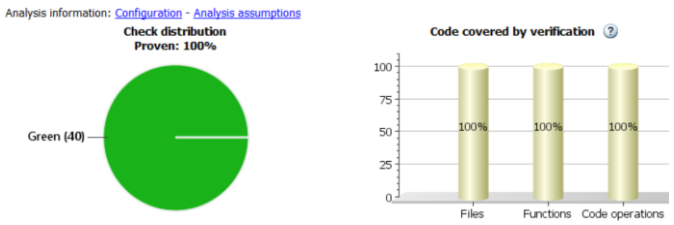

Figure 7: Prerequisite for flow analysis – absence of undefined behavior.

Command executed from externally controlled path (Impact: Medium)
Path to the command argument of 'system' is from an unsecure source.

	Event	File	Scope	Line
1	Formal parameter is a tainted pointer	flows.c	main()	37
2	Assignment to local pointer 'username'	flows.c	main()	38
3	Entering function 'send_alarm'	flows.c	main()	46
4	Call to sprintf	flows.c	send_alarm()	30
5	Entering if branch (if-condition true)	flows.c	send_alarm()	33
6	Command executed from externally controlled path	flows.c	send_alarm()	33

Figure 8: Tainted data flow from external source.

program, we get a confirmation for the absence of any interference (green result in Figure 5). Since the tool is sound, we have successfully verified our objective.

For the second objective, we expect to always have a dependency between `temp` and `calibrated`. We provide and verify the specification "`calibrated` must impact `temp`." The red result in Figure 5 reveals a programming error; the assignment on line 22 has accidentally been commented out. After fixing the issue, the tool finds another–yet intended–dependency on line 40. Hence, we have verified all expected and specified functional dependencies.

C. *Detection of Security Vulnerabilities*

The third objective relates to the cybersecurity properties of our program: Are there any control and data flows that an attacker could exploit?

As a first step, we run a taint analysis. It automatically identifies data controlled by external actors and traces its usage to critical locations in the code. The result (Figure 8) shows that input validation is missing, which could allow a command injection. The user input `username` is not checked and blindly passed (indirectly) to the system call on line 33. Even though we have already limited the inputs with `snprintf`, an attacker could inject additional commands like "`a@b.com; shutdown`". We can eliminate this vulnerability by rejecting all inputs that are not a syntactically valid e-mail address (not shown here for brevity).

Finally, security vulnerabilities can also mean disclosing sensitive data (i.e., a confidentiality breach). In our example, we can consider the username as sensitive data and verify the specification "`username` must not flow into function `logger`". Our analysis falsifies this objective, and the counterexample (Figure 9) shows a trace that allows for writing the username to the logfile.

After the straightforward fix and rerunning the analysis, all three objectives have been successfully verified. Our program is now robust (no undefined behavior), measures the temperature reliably (absence of unwanted flows), and free from common security vulnerabilities.

V. Discussion

The previous examples illustrate that an analysis of data and control flows can help identifying logical defects in the application, well beyond what coding guidelines and functional tests can do. In this section, we discuss the strengths and weaknesses of the described approach and where it is used best.

The presented specification-driven flow analysis is suitable for safety- and security use cases, in particular to support certification and qualification activities for critical software. Since it is sound, the underlying analysis method exhaustively identifies and tracks data and control flows. By design, the analysis may consider a superset of all feasible flows, yet never a subset. This makes it a safe method to demonstrate the *absence* or *independence* of certain data- and control flows, and thereby directly addresses requirements from DO-178C ("Data and Control Coupling [4]), ISO 26262 ("Freedom From Interference" [3]), IEC 62443 ("Restricted Data Flows" [5]) and other safety and security standards.

```
4 |    char button_pressed = read_reg(0xF4);
5 |    if (safety_switch) {
6 |        valve_pos = button_pressed;
7 |    }
```

Figure 10: Is there a definitive flow dependency between `valve_pos` and `button_pressed`?

Conversely, this approach is suboptimal when the objective is to prove the *presence* of a mandatory flow. Consider the situation in Figure 10, where the value of `valve_pos` shall depend on the value of `button_pressed`. The described approach can find a flow dependency between these two variables. However, it can also detect the conditional control dependency on the safety switch. This code permits a scenario where the button never influences the valve position due to the safety switch being permanently false. Only in two specific circumstances exists a simple answer to the verification objective: If the predicate is always true, then the method can confirm the presence of the mandatory data flow. On the other hand, if the condition is always false, then there is no feasible flow, and the method can disprove the expectation. In all other cases, it can only provide a (counter)example and leave it to the reviewer to check whether the conditions therein satisfy the mandatory flow expectation.

A. Threats to Validity

There are several potential systematic errors that may refute the correctness or completeness of the analysis results. However, most of them can be reduced with reasonable effort:

1. **Missing specification:** The analysis can only verify what has been (implicitly or explicitly) specified, and hence cannot verify the correctness of *all* data and control flows. This is a fundamental limitation of all code analysis methods and is not specific to our approach. To reduce the risk of missing specifications, we recommend automatically importing design specifications, such as different safety/security levels and their implied independence.
2. **Incorrect specification:** Errors within the specification itself can result in incorrect outcomes. To mitigate such risks, the analysis tool can detect those that are inconsistent with the code. However, certain classes of misspecification may escape automatic detection due to a lack of understanding of the intention. Implementing a review process for the specifications can help uncover such issues. Additionally, functional testing can serve as a complementary measure.

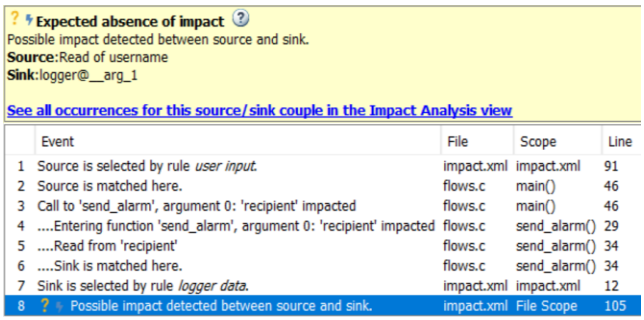

Figure 9: Data flow leaking sensitive data.

3. **Unfixed undefined behavior:** As explained in section I.A., undefined behavior can introduce invisible data and control flows. A sound analysis tool can identify it or prove its absence. By fully removing the identified undefined behavior, users can mitigate this risk.
4. **Incomplete analysis perimeter:** External interactions not included in the analysis–such as unanalyzed processes, assembly code, or the use of libraries where no source code is available–may introduce additional flows. While static code analysis is still effective, it uses "stubs" to represent the missing parts. These are functions without a visible definition and are presumed to have no impact on the known flows. Consequently, a flow could be incorrectly deemed absent when a stub is involved. To mitigate this risk, the tool should warn developers and providing them the opportunity to define the flows through stubs manually. Moreover, the tool should include "smart stubs" for functions from standard libraries to enable sound modeling of the effects of functions like `memcpy`.
5. **Incorrect analysis assumptions:** The correctness of static analysis depends on the analysis setup and assumptions, for example, on the modeled compiler and run-time environment, but also on the range of inputs considered feasible. For a deeper discussion of these assumptions, please refer to [10]. To mitigate the risk of wrong analysis assumptions, we recommend a) using the tool's automatic analysis configuration (e.g., tracing of the build command, compiler detection) and b) minimizing the assumptions, especially regarding ranges and validity of external inputs. This is also beneficial for robustness and cybersecurity properties, as discussed in [10].
6. **Correctness of the analysis tool:** The analysis tool itself is a software and thus might have defects that may lead to wrong outputs. This can be mitigated by a) choosing a sound analysis tool, since it leverages mathematical models that permit no false negatives and b) only using tools that are validated and/or qualified, i.e., checked by certification bodies or validation tests.

The aforementioned tool-related mitigation techniques are implemented in Polyspace Code Prover [8], to reduce systematic errors and to reduce the risk of incomplete results.

VI. Conclusion

Correct data and control flows are crucial for safety and security. Yet, their verification requires going beyond static code analysis. It necessitates checking the flows against a logical specification to detect unwanted interactions within the implementation. Specification-driven flow verification offers a more profound analysis than standard coding guidelines, robustness testing, or requirement-based testing with a debugger.

The described approach can be applied either exploratively, for understanding legacy systems and assessing modifications, or automatically, to enforce a correct-by-design approach and detect logical errors. Our proposed workflow includes (1) extracting expected flows from design specifications, (2) analyzing to confirm or refute these expectations, and (3) examining any discrepancies. Automation is possible for common patterns, such as input validation and SQL injection, while application-specific flows require manual specification.

To obtain correct results and keep the verification process efficient, we suggest (1) eliminating undefined behavior, (2) using capable code analysis tools to verify standard flow patterns, and (3) defining and verifying custom flow specifications. Advanced code analysis tools can support all three steps. Although not covered here, other uses of this methodology include identifying prohibited function calls, analyzing configuration parameters, and ensuring data privacy.

In summary, flow analysis can identify critical defects early in the development process, surpassing traditional robustness and compliance testing and preventing complex bugs during later stages. It helps users verify flow specifications, thereby helping to demonstrate the absence of unintended interferences and side effects.

VII. References

[1] S. McConnell, Code Complete, 2nd Edition ed., Microsoft Press, 2004.

[2] Wikipedia, "Heartbleed," 8 Dec 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Heartbleed&oldid=1185261000.

[3] International Organization for Standardization, "ISO 26262:2018, Road vehicles – Functional safety," 2018.

[4] RTCA, "DO-178C Software Considerations in Airborne Systems and Equipment Certification," 2011.

[5] IEC, "IEC 62443-1-1 Industrial Communication Networks - Network and system Security," 2009.

[6] ISO/IEC, "ISO/IEC 9899:1999 Programming Languages - C," Iso.org, 2011.

[7] The MathWorks, Inc, "Simulink Check," [Online]. Available: https://www.mathworks.com/products/simulink-check.html. [Accessed 01 Dec 2023].

[8] The MathWorks, Inc, "Polyspace Code Prover," [Online]. Available: https://www.mathworks.com/products/polyspace-code-prover.html. [Accessed 01 Dec 2023].

[9] P. Cousot and R. Cousot, "Abstract Interpretation," in *Symposium on Principles of Programming Languages*, 1977.

[10] M. Becker and J. Palczynski, "Increasing Resilience to Cyberattacks through Advanced Use of Static Code Analysis," in *Embedded World Conference*, Nuremberg, 2021.

[11] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering,* Vols. vol. SE-10, no. July 1984, pp. pp.352-357, 1981.

[12] The MITRE Corporation, "Common Attack Pattern Enumeration and Classification (CAPEC)," 2007. [Online]. Available: https://capec.mitre.org/. [Accessed 13 Feb 2023].