A Control-Flow Graph Approach to Language-Agnostic Debugging for Microcontrollers

Carlos Rojas Castillo
Vrije Universiteit Brussel
Brussels, Belgium
carlos.javier.rojas.castillo@vub.be

Matteo Marra
Nokia Bell Labs
Antwerp, Belgium
matteo.marra@nokia-bell-labs.com

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Brussels, Belgium
egonzale@vub.be

Abstract

Language virtual machines (VMs) for resource-constrained environments enabled the use of managed languages, such as JavaScript or Python, on microcontrollers units (MCUs). WebAssembly (Wasm) has also broadened the range of programming languages on these resource-constrained devices. However, most MCU debugging support targets languages that compile to native code, making them unsuitable for source-level debugging of applications running on managed runtimes. As a result, debugging on MCU VMs is often performed using logging, manual resets, and GPIO toggling for call tracing.

In this work, we propose a language-agnostic approach for debugging MCUs. Our approach builds specialised control-flow graphs (CFGs) to enable language-agnostic debugging from compiler-generated Wasm bytecode and debugging information. During debugging, developers can use traditional debugging operations for which the debugger utilises the specialised CFGs to advance computation. We implemented a CFG debugger prototype for the WARDuino Wasm VM, building on a basic debug API. We show that our debugger successfully targets four languages that compile to Wasm without requiring any modification to the debugger. Our benchmarks reveal that the prototype's execution speed outperforms WARDuino's debugger by factors from 7 to 215.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; *Runtime environments*; • **Computer systems organization** → *Embedded and cyber-physical systems.*

Keywords: Debugging, Breakpoints, Stepping, WebAssembly, microcontrollers, IoT

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Carlos Rojas Castillo, Matteo Marra, and Elisa Gonzalez Boix. 2025. A Control-Flow Graph Approach to Language-Agnostic Debugging for Microcontrollers. In . ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Traditionally, programming a microcontroller unit (MCU) was done using languages that compile directly to native code, like C and C++, to meet the performance and hardware constraints of MCUs. Over the past decade, lightweight virtual machines (VMs) have appeared for several mainstream languages, including Espruino [32] and Duktape [62] for JavaScript, MicroPython [28] for Python, and AtomVM [8] for Erlang. Even though those VMs only support a subset of the language's features, they enable the use of managed languages in MCUs, offering benefits such as improved safety, portability, and productivity. WebAssembly (Wasm), a widely adopted bytecode target for many programming languages [6, 14, 41, 44, 81], has reshaped this landscape by offering a portable, efficient, and secure execution model [37]. Wasm VMs like WARDuino [36], WAMR [2] and Wasm3 [46] targeting MCUs have broadened the range of programming languages on these low-power devices from low-level to high-level managed languages.

Debugging programs on MCUs, however, remains challenging. A 2021 study by Makhshari et al. [54] found that programmers need to rely on basic debugging techniques like print statements and manual resets, as debugging support is either missing or not suited for their particular use cases. Two later studies [49, 89] also showed that poorly tested and debugged MCU applications cause bugs in production.

Most of the existing debuggers for MCUs target C/C++ languages [11, 79, 97] compiled to native code. Forks of GDB and LLDB [23, 24] enable debugging beyond C/C++ by leveraging both DWARF debugging information [20] to map native code to source code and OpenOCD [66] to apply machine-level debugging commands (e.g., read registers, write to memory) to the MCU. However, these debuggers require specialised debugging hardware [21] (e.g., JTAG [35]) present or plugged into the MCU, which is not always supported by MCUs (e.g., M5StickC Plus [75]). Moreover, GDB/LLDB debuggers can be used to debug applications running on managed runtimes, but they will debug the VM source code. To debug application-level source code, each VM maintainer needs to implement a debug server and logic to query the CPU state using OpenOCD and convert it into bytecode state. However, this logic is unique per VM, sensitive to changes in the VM software, and requires a tedious, low-level implementation as it needs access to CPU state [66].

In the context of Wasm VMs targeting MCUs, Wasm3 [46] and WAMR [2] cannot debug directly on the MCU nor offer remote debugging of the VM on the MCU. WARDuino [36] does provide remote and out-of-place debugging support [47, 67], but it is limited to AssemblyScript and has experimental support for Rust. To offer online debugging commands, WARDuino's front-end debugger [82] implements step commands by advancing execution one Wasm instruction at a time. This is repeated until a Wasm instruction is found that corresponds to the desired *source location* (i.e., a line and column number in a source file). This results in significantly slower debugging operations, mostly due to the required round-trips between the debugger front-end and the VM running on the MCU.

In this work, we propose a novel *language-agnostic* debugging approach for Wasm VMs targeted to MCUs, enabled through the use of control-flow graphs (CFGs). Concretely, our approach takes as input *unoptimised* compiler-generated Wasm bytecode and debugging information, and creates CFGs specialised for debugging. These CFGs are built before debugging and encode the control flow between all the source locations that can be reached by a debugger. At runtime, a CFG debugger enables debugging operations by searching nodes in these graphs and advancing computation to the Wasm instructions associated with those nodes. The CFGs minimise round-trip messages between the debugger front-end and the VM on the MCU while stepping, as target source locations are derived from the CFGs at the front-end, and sent to the VM, possibly advancing several or many Wasm instructions in one go.

This paper makes the following contributions:

- An algorithm to build CFGs for debugging from Wasm bytecode-level CFGs and standard debugging information (e.g. DWARF). These graphs enable language-agnostic debugging as debugging operations can be implemented by following edges on the graphs, rather than on the syntax and semantics of the source code.
- CFG debugging enables language-agnostic debugging operations for all languages compiling to Wasm. The communication between the debugger front-end and the MCU can be significantly reduced by encoding the source locations within the CFGs.
- A CFG debugger prototype integrated into the WARDuino VM [36]. This integration leverages four existing WARDuino debug API operations. We employed our prototype to debug four different Wasm languages without modifying the prototype itself. Our performance experiments also show that the prototype's debugging operations are 7 to 225 times faster than WARDuino's current debugger.

2 Language-Agnostic Debugging for Wasm Through CFGs

We explore *language-agnostic debugging* for Wasm using control-flow graphs (CFGs). A CFG [1] is a directed graph that represents a program's control flow: the nodes are basic blocks, which contain the longest possible sequence of instructions (e.g., arithmetic operations) without a jump. The last instruction of a block may alter the control by *jumping* to other blocks. In a graph, edges represent control changes.

CFGs are commonly used in static analysis [1, 57] or as an intermediary output during compilation [34, 41, 92]. In our work, we construct CFGs to implement debugging operations such as *step over* by following edges on a graph from one node to another. Before introducing our language-agnostic debugging approach, we introduce a running example used throughout this work.

2.1 Running Example: A Blinking LED Application

Consider a *blinking LED application*, implemented in C, inspired by the work of Yan et al. [98]. In their paper, they examine whether GDB's *step* command correctly advances execution to the next reachable source location. They reported that GDB's *step* command may skip locations when nested loops lack initialisers. Building on this observation, we present an application that blinks an LED and features such a nested loop construct. We later show that our debugger does not skip those locations thanks to the use of control-flow graphs during debugging. Figure 2 shows the main function of the application with the two loops. The full code is in Appendix A.1. The application configures a pin `LED` for output (line 23) and, over two iterations (line 29), continuously increments a variable `delta` (line 30), turns the LED on (line 31), sleeps (line 32), turns the LED off (line 33), and sleeps (line 34). The incremented `delta` ensures that the total sleep time reduces at each iteration.

2.2 Overall Architecture

Figure 1 shows the overview of language-agnostic debugging of MCUs. Before debugging (left Figure 1), we assume the developer compiled a source program with debugging flags on, resulting in an *unoptimised Wasm module* along with its *debugging information*. The program may have been written in traditionally native-compiled languages (e.g., Rust, C, Go) or managed languages (e.g., AssemblyScript). The debugging information is expected to adhere to standards like DWARF [20] or Source Map Specification [73]. Both are widely used to debug desktop applications through tools like GDB/LLDB [29, 50] and TypeScript/JavaScript applications running on browsers [30, 58, 61].

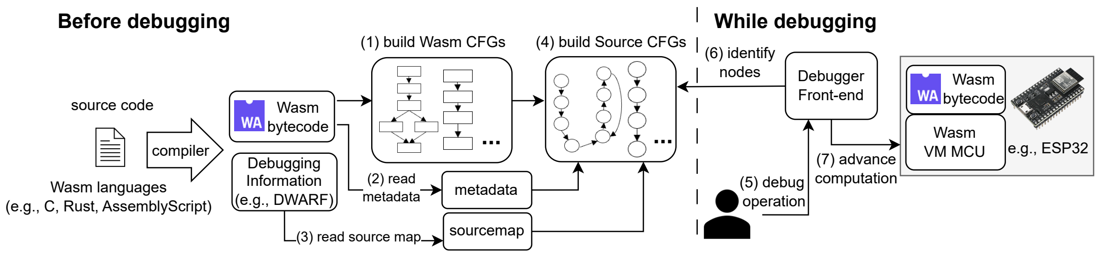
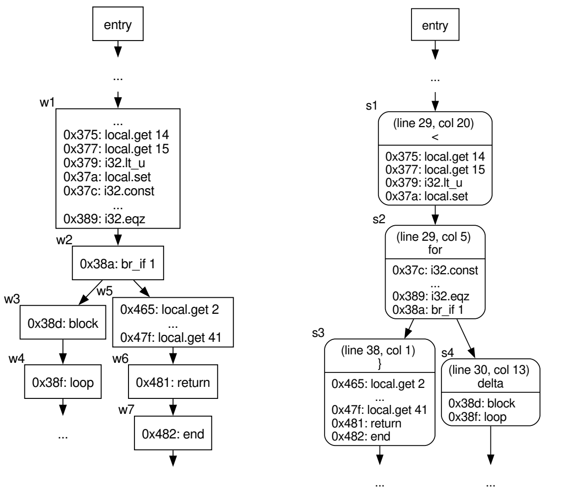


Figure 1. Overview of a CFG-Based debugging session.

```
18 int main() {
19     unsigned int LED = 10;
20     unsigned int OUTPUT = 2;
21     unsigned int ON = 1;
22     unsigned int OFF = 0;
23     pin_mode(LED, OUTPUT);
24
25     unsigned int SLEEP = 2000;
26     unsigned int TOTAL_RUNS = 2;
27     int run_idx=0;
28     unsigned int delta = 0;
29     for (; run_idx < TOTAL_RUNS; run_idx++){
30       for(; delta < SLEEP; delta +=100 ){
31         digital_write(LED, ON);
32         delay(SLEEP - delta);
33         digital_write(LED, OFF);
34         delay(SLEEP - delta);
35       }
36       delta = 0;
37     }
38 }
```

Figure 2. The main function of a C application that for two runs blink an LED at a gradually increasing frequency (full code in Appendix A.1).



(a) WCFG extract of the *main*. **(b)** SCFG extract of the *main*

Figure 3. The WCFG and its corresponding SCFG, constructed for the main function in Figure 2. *entry* marks the start of the CFG, dots indicate omitted nodes and instructions. For clarity, source code is included in the SCFG nodes.

To debug the program on an MCU, such as on the popular ESP32[1] (right in Figure 1), the developer interacts with a front-end debugger integrated into an editor (e.g., VSCode). When launched, the debugger front-end uploads a copy of the Wasm module *without* debugging information to the Wasm VM running on the MCU. In parallel, the debugger builds two types of CFGs on the developer's machine:

- A *Wasm Bytecode-Level Control-Flow Graph* (WCFG) for each Wasm function defined in the module (1). A WCFG depicts the control flow of Wasm bytecode instructions. To illustrate such a graph, consider Figure 3a, which shows the WCFG of the main function from the blinking LED application (shown in Figure 2) and highlights nodes corresponding to lines 29, 30, and 38. Each WCFG node is a basic block containing Wasm instructions that run in increasing order (e.g., from *0x465* to *0x47f* in *w5*). WCFG edges indicate how control flows between nodes (e.g., instruction *0x481* in *w6* is executed after *0x47f* in *w5*). To build WCFGs, we employ the same approach as Stiévenart et al. [77], where a WCFG for a Wasm function is built by traversing the function's body, collecting Wasm instructions that belong to a block [94] into a node, and adding edges between those nodes when encountering Wasm branching instructions (e.g., br, br_if).
- Using metadata (2) extracted from the module (e.g., function identifiers[2], type signatures), and a *sourcemap*[3] (3) extracted from the debugging information, a *Source-Level Control-Flow Graph* (SCFG) is constructed for each WCFG (4). An SCFG depicts the control flow of a Wasm function at the source code level. Figure 3b

[1]According to a 2023 report, Espressif, the manufacturer of the ESP32, reported over 1 billion sales worldwide https://www.espressif.com/en/news/1_Billion_Chip_Sales.

[2]In Wasm, each function has a corresponding identifier.
[3]In DWARF [20], the sourcemap is called *debug lines*.

depicts the SCFG for the WCFG shown in Figure 3a, corresponding to our running example. Each SCFG node contains a *source location*, i.e., line and column number in a source file, the Wasm instructions associated with the source location, and possible function identifiers that this node may call. For instance, node *s1* (Figure 3b) points to (line 29, col 30) and contains instructions of $w1$. Edges represent control flow between source locations.

The construction of the WCFGs, sourcemap, and metadata can be done in parallel. Using the Wasm module and debugging information, multiple SCFGs get created. Once the SCFGs are created, the debugger is ready for use.

While debugging (right Figure 1), the developer issues debugging operations (5). Per operation, the debugger front-end uses the SCFGs to identify nodes (6) that, when reached, would correspond to the completion of the requested debugging operation. For instance, consider the scenario where the application is paused at (line 29, col 5), which corresponds to the Wasm instruction with address $0x37c$ in node $w1$ (Figure 3a). If a *step* is applied, the debugger identifies SCFG node $s2$ in Figure 3b as the one corresponding to address $0x37c$, and nodes $s3$ and $s4$ as the locations where computation could advance. Using the Wasm instructions associated with the identified nodes, the debugger (7) controls the application running on the MCU by setting breakpoints at the addresses of the identified nodes (e.g., $0x465$ for $s3$ and $0x38d$ for $s4$). The operation completes once execution reaches one of these breakpoints. Stepping from $s2$ to $s3$ or $s4$ highlights how CFG debugging does not suffer from the issue Yan et al. [98] reported with GDB step command for nested loops lacking initialisers, as it does not bypass line 30.

3 Building Source-Level Control-Flow Graphs (SCFGs)

We now introduce our algorithm for building SCFGs from WCFGs, debugging information, and metadata. Assuming WCFGs have been built from the Wasm bytecode, to build SCFGs, we first need to extract metadata from the Wasm bytecode. This metadata contains all the Wasm functions and their type signature identifiers[4], which we write as set $\mathcal{F} = \{(f, t) \mid f \text{ is a function ID and } t \text{ a type signature ID}\}$. Second, we extract the *sourcemap* from the debugging information, which is a function that maps the *address* of a Wasm instruction to a *source location*:

$$sourcemap(addr) = \begin{cases} (s, l, c) & \text{source location} \\ \emptyset & \text{no mapping found} \end{cases} \qquad (sm)$$

where l is line number, c is column number, in source file s

Using the WCFGs, the *sourcemap*, and $\mathcal{F}$, the debugger builds an SCFG for each WCFG, illustrated in Algorithm 1.

[4]In Wasm, each type signature has an associated identifier.

Algorithm 1: Build SCFGs from WCFGs.

```
Input: Set of WCFGs, set of function IDs with their type
       signatures $\mathcal{F}$, a sourcemap as defined in (sm)
1  . Output: Set of constructed SCFGs
2  $\mathcal{SCFGS} \leftarrow \emptyset$          ▷ Set of constructed SCFGs
3  foreach $\mathcal{W} \in \mathcal{WCFGS}$ do
4      $\mathcal{N}, \mathcal{E} \leftarrow \emptyset$          ▷ SCFG nodes and edges
5      $buildNodes(\mathcal{W}, \mathcal{N}, \mathcal{E}, \mathcal{F}, sourcemap)$     ▷ Pass 1
6      $buildEdges(\mathcal{W}, \mathcal{N}, \mathcal{E})$                ▷ Pass 2
7      $mergeNeighbours(\mathcal{N}, \mathcal{E})$                ▷ Pass 3
8      $\mathcal{N}_e \leftarrow identifyEntryNodes(\mathcal{W}, \mathcal{N}, \mathcal{E})$
9      $\mathcal{SCFGS} \leftarrow \mathcal{SCFGS} \cup \{(\mathcal{N}_e, \mathcal{N}, \mathcal{E})\}$
10 return $\mathcal{SCFGS}$
```

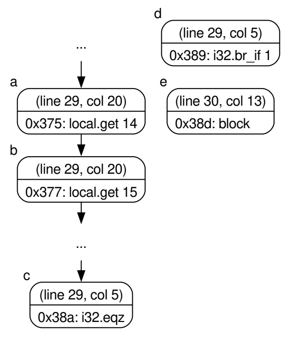
For each WCFG $\mathcal{W}$, the algorithm creates $\mathcal{N}$ and $\mathcal{E}$, two empty sets which will eventually contain the constructed SCFG nodes and edges (line 4). The algorithm then constructs the SCFG in three different passes. To illustrate these passes, consider Figure 4, which shows the intermediary SCFGs produced by the first two passes, applied on a subpart of the WCFG (Figure 3a). The final SCFG, resulting from the third and final pass, was previously shown in Figure 3b. Below, we first provide a conceptual overview of each pass, followed by detailed explanations in the subsequent subsections.

The first pass (line 5) creates the SCFG nodes $\mathcal{N}$ and possibly SCFG edges $\mathcal{E}$ using a WCFG, the *sourcemap*, and $\mathcal{F}$. For instance, when applied to $w1$, $w2$, and $w3$ of the WCFG (Figure 3a), three subgraphs are produced as shown in Figure 4a. Pass 1 also identifies the Wasm functions that an SCFG node may call. We call such nodes *call nodes*. Identifying call nodes is important for enabling debugging operations. As each wasm function has its own CFG, calls to another function need to properly be identified to point to the right CFG.

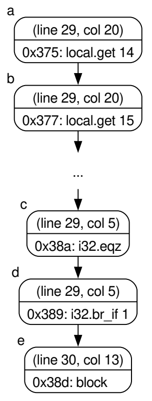
The second pass (line 6) adds edges to the SCFG. For each WCFG edge (w_{from}, w_{to}), it identifies corresponding SCFG nodes s_{from} and s_{to}, and connects them. For instance, in Figure 4b, edges (c, d) and (d, e) were added after pass 1. If w_{to} has no corresponding SCFG node (due to missing source locations in the *sourcemap*), the pass connects s_{from} to all the closest SCFG nodes reachable from w_{to} [5]. This ensures that the control following s_{from} is included in the SCFG. By the end of this pass, all subgraphs from pass 1 are connected.

The third pass serves two purposes. First, it fixes a granularity issue (line 7) caused by the first two passes: each SCFG node contains only one Wasm instruction, causing the debugger to step instruction by instruction. For instance, advancing from *a* to *b* (Figure 4b) runs one Wasm instruction, rather than the full sequence leading to *c*. Second, the pass identifies the SCFG entry nodes (line 8). The obtained nodes, edges, and entry nodes form the SCFG (line 9).

[5] w_{from} may also have no corresponding SCFG node, but this gets automatically resolved by pass 2.



(a) Pass 1



(b) Pass 2

Figure 4. Left: The SCFG after pass 1 applied to $w1$, $w2$, and $w3$ (Figure 3a). Right: The SCFG after pass 2 applied on edges $(w1, w2)$ and $(w2, w3)$. Dots indicate omitted nodes.

Algorithm 2: Pass 1 - Creating SCFG nodes.

Input : A WCFG $\mathcal{W}$, SCFG nodes $\mathcal{N}$, SCFG edges $\mathcal{E}$, set of function IDs with their type signatures $\mathcal{F}$ and *sourcemap* as defined in (*sm*).

Output : $\mathcal{N}$ and $\mathcal{E}$ respectively extended with nodes and edges.

```
1  foreach $w \in breadthFirstTraverse(\mathcal{W})$ do
2      $sn_{prev} \leftarrow \bot$        ▷ previously created SCFG node
3      foreach $i \in instructions(w)$ do
4          if $sourcemap(i) \neq \emptyset$ then
5              $(s, l, c) \leftarrow sourcemap(i)$
6              if $isFileAvailable(s)$ then
7                  $fids \leftarrow calls(i, \mathcal{F})$
8                  $sn \leftarrow ((s, l, c), \{i\}, fids)$
9                  $\mathcal{S} \leftarrow \mathcal{S} \cup \{sn\}$
10                 if $sn_{prev} \neq \bot$ then
11                     $\mathcal{E} \leftarrow \mathcal{E} \cup \{(sn_{prev}, sn, i)\}$
12                 $sn_{prev} \leftarrow sn$
```

3.1 Pass 1: Creating SCFG Nodes

Algorithm 2 outlines the steps to create the SCFG nodes and possibly edges, as well as to identify the call nodes. The algorithm breadth-first traverses the WCFG. For each node w, it visits the w's instructions in *ascending* address order (line 3). For each instruction i, it checks if i has a valid source location, i.e., the address of i maps to a *source location* (line 4) and source file s is present on the local machine. The latter is tested using `isFileAvailable` (line 6). Occasionally, no source location is found for i because it may have been omitted by the compiler. In a such case, our algorithm does not create an SCFG node. This implies that our debugger, similarly to other debuggers [29, 50], cannot show source-level information for i.

When i has a valid source location, the algorithm checks whether i is a call instruction. If so, it computes the set of function identifiers $fids$ that could be called by i. For this, we use helper function *calls*, that given i and $\mathcal{F}$, extracts function identifiers depending on i (line 7):

- If i is a *direct call* instruction, the identity of the called function is encoded in the bytecode. By reading the bytecode, we identify which function is called. This identifier is returned as a result.
- If i is an *indirect call* instruction, the identity of the called function might only be known at runtime. Regardless of which function i calls, only functions with the same type signature as i can be called. Since this type signature is known statically, we read it and return all the function identifiers in $\mathcal{F}$ that have a matching type signature. The returned set includes all the functions that could be called by i.
- Else, return an empty set since i is not a call instruction.

The algorithm then creates the SCFG node sn containing the source location, i, and $fids$ (line 8). For instance, node a (Figure 4a) is created for i with address $0x375$ of node $w1$ (Figure 3a). Debugging operations can determine if sn is a *call node* by checking whether the node's $fids$ are not empty.

After each iteration, we keep track of the last generated SCFG node sn_{prev} (line 12) and add an edge from sn_{prev} to sn (line 11). In Figure 4a, this caused edges to be added between nodes a to c. Each edge stores instruction i to indicate that after executing the instructions of sn_{prev}, control flows to i of sn. Including i in the edge is important since multiple SCFG nodes may branch to different instructions belonging to the same SCFG node. The instruction i enables the debugger to accurately set breakpoints among these instructions. Moreover, adding the (sn_{prev}, sn) edge is necessary to build correct execution paths. Since the instructions of w run in sequence and the instructions of sn_{prev} run before sn. The added edge preserves the control flow of w in the SCFG.

3.2 Pass 2: Adding SCFG Edges

Algorithm 3 outlines the steps to add the SCFG edges. The algorithm traverses a given WCFG (line 1 Algorithm 3) and adds an SCFG edge for each WCFG edge (w_{from}, w_{to}). Conceptually, this involves three steps, which we now describe.

Step 1: Identify sn_{from} associated with w_{from} (line 2 Algorithm 3). An edge from w_{from} to w_{to} means that after running the highest address instruction of w_{from}, control flows to the lowest address instruction of w_{to}. Therefore, adding the SCFG edge for (w_{from}, w_{to}) requires identifying the sn_{from} containing the highest address instruction of w_{from}. This is enabled by helper function *highestAddrNode* (line 2 Algorithm 3) that iterates over w_{from}'s instructions in

Algorithm 3: Pass 2 - Adding SCFG edges.

```
Input  : A WCFG $\mathcal{W}$, SCFG nodes $\mathcal{N}$, and SCFG edges $\mathcal{E}$
Output : $\mathcal{E}$ extended with the edges
1 foreach $w_{from} \in breadthFirstTraverse(\mathcal{W})$ do
2     $sn_{from} \leftarrow highestAddrNode(w_{from}, \mathcal{N})$   ▷ SCFG node with the highest address
3     if $sn_{from} \neq \perp$ then
4         foreach $w_{to} \in neighbours(w_{from})$ do
5             foreach $(sn_{to}, i_{to}) \in$ closestNodes$(w_{to}, \mathcal{N}, \mathcal{SE})$ do
6                 $\mathcal{E} \leftarrow \mathcal{E} \cup \{(sn_{\text{from}}, sn_{to}, i_{to})\}$
```

decreasing order and returns the first found SCFG node. For instance, for $w1$ (Figure 3a), *highestAddrNode* identifies c (Figure 4a) as it corresponds to the instruction with the highest address in $w1$ that has an associated SCFG node. If none of $w1$'s instructions have a SCFG node, *highestAddrNode* returns *nothing* and Algorithm 3 proceeds to the next node.

Step 2: Identify the SCFG node sn_{to} or the closest SCFG nodes to w_{to} (lines 4-5 Algorithm 3). After identifying sn_{from}, Algorithm 3 searches per edge (w_{from}, w_{to}), the SCFG node sn_{to} that corresponds to w_{to} (lines 4-5). It could be that none of the w_{to}'s instructions have a corresponding SCFG node. In that case, the closest SCFG nodes to w_{to} are then searched. Algorithm 4 `closestNodes` handles both cases: whether w_{to} has a SCFG node or not.

In case w_{to} has a corresponding SCFG node, *closestsNodes* finds sn_{to} using helper function *lowestAddrNode* (line 1 Algorithm 4). The function iterates over w_{to}'s instructions in ascending order until it finds an instruction i associated with an SCFG node. That node is then returned from `closestNodes` (line 8). For instance, if w_{to} is $w2$ (Figure 3a), `closestNodes` identifies d (Figure 4a) since it is the associated SCFG node to the first instruction of $w2$. Searching in increasing order follows the Wasm control: after executing w_{from}'s instructions, the instructions of w_{to} are executed in ascending order.

Algorithm 4: Find the closest SCFG neighbours of WCFG node w.

```
Input  : WCFG node $w$, SCFG nodes $\mathcal{N}$, and SCFG edges $\mathcal{E}$
Output : Set of $(sn, i)$ where $sn \in N$ and $i$ is a Wasm instruction
1 $(sn, i) \leftarrow lowestAddrNode(w, \mathcal{N})$
2 if $sn = \perp$ then
3     $\mathcal{A} \leftarrow \emptyset$
4     foreach $w_{nb} \in neighbours(w)$ do
5         $\mathcal{A} \leftarrow \mathcal{A} \cup closestNodes(w_{nb}, \mathcal{N}, \mathcal{E})$
6     return $\mathcal{A}$
7 else
8     return $\{(sn, i)\}$
```

In case w_{to} has no SCFG node, `closestNodes` searches for its closest (direct) SCFG nodes. To this end, `closestNodes` recursively calls itself on each outgoing neighbour w_{nb} of w_{to} (lines 4-5). If w_{nb} has an associated SCFG node, it is returned. Otherwise, the search proceeds to the outgoing neighbours of w_{nb}. The algorithm returns a set of (sn, i) representing the SCFG nodes and instructions associated with the neighbours.

Step 3: Add an edge between sn_{from} and sn_{to} (line 6 Algorithm 3). The final step is to add an edge from sn_{from} to either sn_{to} or the closest neighbours of w_{to} while storing instruction i that leads to them (Algorithm 3 line 6). For instance, for edges $(w1, w2)$ and $(w2, w3)$, an SCFG edge is added respectively between (c, d) and (d, e) as illustrated in Figure 4b. Adding the edges gradually builds the SCFG execution paths: either an edge (sn_{from}, sn_{to}) is added to mirror an existing (w_{from}, w_{to}) edge, or edges are added to neighbours of w_{to} that do have a SCFG node, thus preserving the control flow that follows after executing w_{to}. In both cases, the algorithm ensures that all execution paths are part of the SCFG.

3.3 Pass 3: Readying the SCFG for Debugging

The goal of the final pass is (1) to fix the node granularity of the SCFGs, and (2) to identify the SCFG entry nodes. Due to the page limit, we present only a conceptual overview of this pass rather than a detailed algorithm.

Merging Neighbouring Nodes. To fix the node granularity, the third pass merges neighbouring nodes that point to the same source location. This is achieved by traversing the SCFG using *breadth-first traverse*. For each sn node, we determine whether sn can be merged with its neighbouring nodes by checking whether all the incoming edges to sn originate from nodes with the same source location as sn. If so, sn and its neighbours are merged into a single SCFG node sn_{merged}. Node sn_{merged} retains all incoming and outgoing edges from the original nodes. If one incoming SCFG node points to a different source location than sn, nothing gets merged. This pass yields a more compact SCFG by shortening execution paths. Applying this pass on the SCFG of Figure 4b, results in the SCFG illustrated in Figure 3b.

Identifying SCFGs Entry nodes. After merging the nodes, the final step is to find the SCFG nodes that correspond to the entry nodes. This is done by applying *closestNodes* (Algorithm 4) on the entry node w_{entry} of the WCFG used to create the SCFG. If w_{entry} has a corresponding SCFG node, that node becomes the entry node for the SCFG. If w_{entry} has no corresponding SCFG node, the closest (direct) SCFG nodes to w_{entry} become the entry nodes.

4 Implementing a CFG Debugger

We implemented a CFG debugger prototype for the WARDuino [36] Wasm MCU VM. The debugger front-end was

```
1  function stepInto(scfgs,sn){
2   if (isCall(sn)) {
3   const fs = funcs(sn)
4   return entries(scfgs,fs)
5   }
6   return stepOver(scfgs,sn)
7  }
8  function startNode(scfgs,rt){
9   const addr=rt.inspectPC()
10  for(const n of scfgs.nodes)
11   for(const i in n.insts)
12    if (i.addr === addr)
13      return n
14 }
15 function advance(ds,rt){
16  const as=ds.map(n=>n.addr)
17  for (const a of as)
18   rt.addBreakpoint(a)
19  rt.run()
20  rt.onBPReached(()=>{
21  for (const a of as)
22   rt.removeBreakpoint(a)})}
```

Figure 5. JavaScript-like pseudo-code of a CFG step into. The arguments `scfgs` and `rt` represent, respectively, all SCFGs and the runtime that runs the target Wasm.

built by extending WARDuino's VSCode debugger client extension [82]. The obtained extension builds and uses SCFGs, as discussed in Sections 2 and 3, replacing WARDuino's original stepping approach to debugging operations. For the implementation, we also used libraries WebAssembly-js [68], wasm-tools [3], and source-map [59] to respectively parse a Wasm module and extract a *sourcemap* from DWARF or Source Map Spec. The following describes the runtime debugging API, the implementation of CFG debugging operations, and how a CFG debugger handles MCU interrupts.

4.1 The Wasm Runtime Debugging API

A CFG debugger advances computation from one Wasm address to another, each associated with specific SCFG nodes. It achieves this by setting and removing breakpoints at these Wasm addresses. To support this functionality, a CFG debugger expects the runtime to expose the following basic debugging API for controlling the deployed Wasm module:

addBreakpoint address pause Wasm execution at the provided Wasm address.

removeBreakpoint address remove the breakpoint at the Wasm address.

run resume execution of the Wasm module.

inspectPC return the program counter, i.e., the current Wasm address being executed.

This API consists of commonly supported operations and is already implemented in the WARDuino VM, allowing us to integrate our prototype without changing the VM.

4.2 Implementing CFG Stepping Operations

Implementing *stepping debugging operations*, such as *step into*, *step over*, and *step out* in a CFG debugger, involves combining calls to the runtime debugging API with operations upon the SCFGs. In the following, we explain how to implement step into, step over, and step out. Due to the page limit, we only provide a detailed explanation for step into. Step over and step out are described conceptually.

Step Into. This operation steps into a function call and pauses at its entry. To this end, the debugger calls the three functions illustrated in Figure 5. First, the debugger identifies the SCFG node corresponding to the current paused source location, which we call the *start node*. For this, it calls the `startNode` function (line 8), passing all SCFGs and runtime as arguments. `startNode` reads the runtime PC (line 9), which points to a paused Wasm instruction, and iterates through all SCFG nodes and instructions to find the node matching the inspected PC (lines 10-13).

Second, the debugger identifies the SCFG node(s) to where execution should advance; we call these nodes *destination nodes*. To find the destination nodes, the debugger calls `stepInto` with the set of SCFGs and the start node `sn` (line 1). The function checks if `sn` is a call node (line 2). If so, step into gets the function identifiers that could be called using `funcs` (line 3). Then, it retrieves the SCFGs and their entry nodes for each function using `entries` (line 4). These entry nodes are the destination nodes (line 5). If `sn` is not a call node, step into executes a step over (line 6).

Third, the debugger advances the computation of the runtime to the destination nodes by calling the helper function `advance` (line 15) with the destination nodes and runtime as arguments. This function retrieves the start address of each destination node (line 16), sets breakpoints at those addresses (lines 17-18), and resumes execution of the Wasm module (line 19). When a breakpoint is hit, it removes all previously set breakpoints (lines 20-22), ending the step into.

Step Over. This operation is applied upon a function call to advance computation until the call completes. To perform a step over, the debugger retrieves the start node using `startNode`. Then the debugger accesses the destination nodes, which are the neighbours of the start node (code in Appendix B). Regardless of whether the start node is a call node, a neighbour represents the location where control flows after completing the call. If no neighbour exists, the current call has completed, and step over executes a step out. Finally, the debugger advances to the destination nodes using `advance`.

Step Out. Step out resumes the execution of the current function until it completes, pausing at the first source location reached after the call returns. To enable step out, the debugger first identifies the start node using `startNode`. Then the debugger finds the destination nodes (code in Appendix B). For this, it iterates over all the SCFGs to find all the call nodes that could call the current function. Then it applies a step over on each call node. Ideally, the debugger only steps over the call node *n* that called the current function, but since it is not possible to statically identify *n*, the debugger conservatively targets all call nodes, ensuring the correct call node is included as destination node. Lastly, the debugger advances to the destination nodes.

4.3 Pausing Interrupt Handlers

MCU developers can implement *interrupt service routines* (ISR) [39] functions and register them to be executed on interrupts, such as a pin toggled from a button press. When an interrupt occurs, the MCU OS temporarily pauses the `main`'s execution and runs the ISR until completion. To debug user-defined ISRs, the MCU runtime needs logic to catch and handle interrupts. With CFG debugging, this task can be simplified by statically identifying the SCFGs of the ISRs and setting breakpoints at their entry nodes.

To identify these SCFGs, we assume user-defined ISRs are not (directly or indirectly) called from main. This is a fair assumption since popular MCU OSs [26, 101] and programming frameworks [4, 22] follow this pattern. Thus, identifying ISRs involves finding functions not reachable from main. For this, the debugger tracks which functions are called by the call nodes starting from the `main`'s SCFG. For each function f called by the `main`'s call nodes, we access f's SCFG and its call nodes to identify the functions called by f. We recursively repeat this process for each function called by f. Functions that are not encountered while following the call nodes are the interrupt handlers.

While this approach detects ISRs not called from main, it may incorrectly flag unrelated functions (e.g., an unreachable function never registered as an ISR). To fix this issue, the debugger could let the developer manually remove these false positives via the GUI, for instance.

5 Evaluation

We evaluated the practicality and effectiveness of our approach by conducting several experiments to answer the following research questions:

RQ1. Can we build SCFGs for realistic Wasm applications in a language-agnostic manner?
RQ2. Are the SCFGs complete with respect to execution paths and call nodes?
RQ3. Does CFG debugging have comparable performance to state-of-the-art Wasm MCU VM debugging?
RQ4. Can a CFG debugger be extended to target new programming languages that compile to WebAssembly?

Experimental Setup. To address those questions, we conducted several benchmarks. Each benchmark was executed using *unoptimised Wasm modules* and a machine[6] equipped with 32 GB RAM, 10 CPU cores, 1 TB SSD storage, OS Sequoia 15.5, and Darwin Kernel Version 24.5.0. For MCU-related benchmarks, we used the WARDuino VM commit *3e97e4f* deployed on an M5StickC ESP32 board [75] that has 520 KB of RAM and 4 MB of flash memory.

[6]https://support.apple.com/en-us/111902 (Accessed on 24/08/2025)

5.1 RQ1: Building SCFGs for Realistic Wasm Applications

We now evaluate whether our proposed algorithm (Section 3) can construct SCFGs for realistic Wasm applications. To this end, we first show that we can build WCFGs for arbitrary Wasm modules. Since any language generating Wasm must adhere to the official Wasm specification [94], our initial goal is to show that WCFGs can be generated for the entirety of the WebAssembly specification. Following this, we show that SCFGs can be built in a language-agnostic manner for WebAssembly projects written in different languages.

5.1.1 Experiment 1: Covering the Wasm Specification.

We examine whether our approach can build WCFGs for modules that satisfy the WebAssembly version 1 specification, as this is currently the standard WebAssembly version. While there are extensions to the core specification[7], such as garbage collection and SIMD, these are still in development. Therefore, we do not consider them in our evaluation.

We used the official WebAssembly test suite [93], which provides 147 Wast (Textual WebAssembly)[8] modules designed to test compliance with the Wasm specification and ongoing proposals. From these tests, we selected the 76 modules that test compliance with Wasm version 1.

Out of the 76 Wast modules, we successfully constructed WCFGs for 75 modules and encountered an exception on one module caused by WebAssembly-js [68], a library that we use for parsing Wasm modules. Specifically, the library failed to properly decode an *int32* value from the module, considered too large to fit into an *int32* buffer. Despite this exception, we found that the complete set of Wasm version 1 instructions is used by the other Wast modules for which WCFGs were successfully constructed. As a result, our approach was tested on all the Wasm version 1 instructions. We conclude that our approach can construct WCFGs for a broad range of Wasm modules that adhere to the Wasm version 1 specification.

5.1.2 Experiment 2: Building SCFGs.

In this experiment, we determine whether our approach can construct SCFGs for all the projects listed in Table 1. We collected those projects from publicly available Rust and AssemblyScript (AS) projects. The selected projects encompass benchmarks (△), libraries (★) and real-world applications (□). The benchmarks (△) consist of the Rust and AS's official compiler test suites [7, 80] and specialised benchmarks on rendering and computational performance measurement [5, 15, 88]. The libraries (★) domains include image processing [70], blockchain [12], data conversion [18], crypotography [17], and database [85]. Finally, the real-world applications (□)

[7]https://webassembly.org/features/ (Accessed on 24/08/2025)
[8]Wast is a human-readable textual representation of a Wasm module that has a one-to-one mapping with its compiled Wasm bytecode module.

are in the domain of blockchain, terminal prompts, and emulators [12, 78, 83]. Projects also vary in the total number of lines of code (LoC).

Table 1. Rust and AS projects that compile to Wasm. The projects are libraries (★), benchmark suites (△), real-world applications (◻), or a combination of these. The *AS* and *Rust compiler* correspond to the official AS and Rust test suites used by the language maintainers.

Lang	Project	LoC	Built	Comp.
AS	as-bind [84]★	129	100%	23
	as-benchmark [88]△	603	100%	28
	wasm-mandelbrot [15]△	42	100%	4
	as2d [5]△	374	100%	5
	AS compiler [7]△	12006	99.24%	608
	wasm-crypto [17]★◻	1463	100%	4
	wasmboy [83]★△◻	6347	100%	138
				Total compares: 810
Rust	limbo [85]★	1589	100%	3957
	photon [70]★	2959	100%	61612
	excel2json [18]★	824	100%	3815
	Rust compiler [80]△	66689	100%	23818
	cep18 [12]★◻	827	100%	12947
	genact [78]◻	960	100%	17225
				Total compares: 123374

We compiled each project to Wasm *without optimisation* enabled, while including the associated debugging information: DWARF [20] for Rust and Source Map Spec [73] for AS. Using the obtained modules and the *sourcemap* extracted from the debugging information, we constructed SCFGs using our prototype. In the results of this experiment, we only include the modules for which a valid *sourcemap* was produced. Modules with either no *sourcemap* or incorrect mappings (e.g., mappings containing nonexistent Wasm addresses) are excluded. This is because for these modules, either the SCFG cannot be produced due to the lack of a *sourcemap*, or the produced SCFG is incorrect due to incorrect mappings produced by compilers [98].

The *built* column in Table 1 indicates the percentage of successfully used Wasm modules to construct SCFGs. The results show that SCFGs could be constructed for all but one Wasm module part of the AS compiler suite. The module raised a parsing error in a dependent library WebAssembly-js [68], which failed to decode a portion of the module. Since the module is reported as valid [7], the error suggests a bug present in the library. Consequently, the success rate for the AS compiler suite is slightly lower at 99.24%.

We conclude that our approach can build SCFGs solely from unoptimised Wasm modules and a *sourcemap* extracted from debugging information. As the modules originate from different languages, producing different debugging standards, the SCFG construction can be applied to other programming languages compiling to Wasm.

5.2 RQ2: Are the SCFGs Complete

To evaluate if the constructed SCFGs are *complete*, two conditions must be met. First, each SCFG built for a function contains all the execution paths of the function. Second, the call nodes of SCFGs contain all the functions that could be called at runtime. We conducted two experiments to validate those two conditions.

5.2.1 Experiment 1: Examining the Execution Paths.

To verify that SCFGs contain all execution paths, we compare the SCFGs to the CFGs produced by the compiler. These compiler-generated CFGs (CCFGs) are derived from source code and reflect control flow at the source level [14, 41]. Since a CCFG contains all execution paths, comparing it to its corresponding SCFG allows us to determine whether the SCFG contains all the execution paths. This assumption is valid since the SCFGs are produced from unoptimised Wasm, which preserves the original control flow of the program. As SCFGs differ from CCFGs (e.g., while a SCFG node contains both bytecode and source-level information, a CCFG node only contains source-level information), we compare both graphs on *structural similarity* [40]. This comparison tells us whether both graphs have the same execution paths.

In literature, structural similarity has largely been studied and used for malware [25, 27, 40, 52] and plagarism detection [69]. To measure structural similarity, we use CFG comparison algorithms [13, 40] that compute a similarity value $s \in [0, 1]$ for two CFGs based on their structural similarity: the higher s, the more structurally similar the graphs. While there exist different algorithms to compute s (e.g., k-subgraphs, edit distance) [40], no consensus exists on which to pick [13, 69]. We employ a Python tool [13] that implements the *simulation-based* algorithm introduced by Sokolsky et al. [71], as it is well-suited for the characteristics of SCFGs. First, the algorithm is robust to path length differences, thus suited for comparing longer SCFG paths to shorter CCFGs caused by their granularity differences. The difference in granularity stems from our method, which constructs each SCFG node to represent a single source location. In contrast, CCFG nodes generally represent a statement that may cover multiple source locations. The recursive nature of the algorithm guarantees that SCFG nodes situated deeper are compared to CCFG nodes positioned higher. Second, the algorithm does not require the node's internal information and edge labels to compute s. This is good since SCFGs lack edge labels and have different node content than CCFGs.

We conduct this experiment on all projects listed in Table 1 by running two steps:

1. *Create CCFGs.* For Rust, we configured the compiler to produce MIR (MID Level IR). For AS, we used a

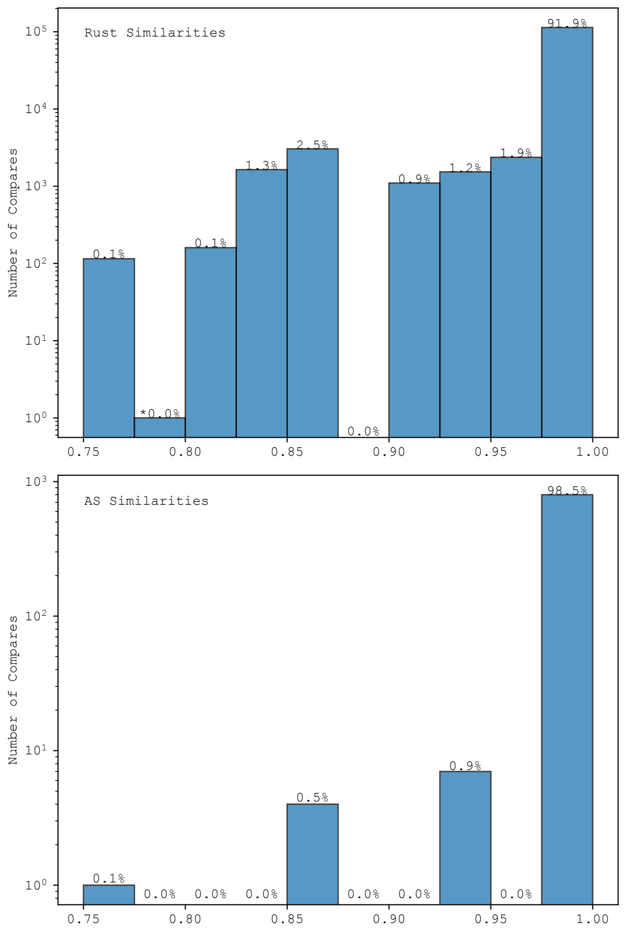


Figure 6. Histograms relating similarity ranges (x-axis) with a width of 0.025 and their frequency of occurrence (y-axis) plotted on a logarithmic scale. Each bar label indicates the percentage of comparisons that fall within the range. The ∗0.0% on the Rust diagram is not zero; rather, the true percentage is so small that it rounds down to zero.

JavaScript-CFG [99] (as the AS compiler does not produce source-level CCFGs) that we slightly transform to ease the next step.

2. *Compute similarities.* Access each SCFG and its corresponding CCFG. Then, compute a similarity value using the CFG comparison Python tool [13].

We obtained similarity values for each (*SCFG*, *CCFG*) pair. Column *Comp.* of Table 1 indicates the number of comparisons executed per project. The table also shows the comparison per language: we performed 810 comparisons for AS and 123374 for Rust. Since some projects have a low number of LoC, we grouped the similarity results by language and then computed statistical metrics, as this ensures that results are representative. Figure 6 displays histograms relating similarity ranges to their frequency expressed on a logarithmic scale. The histograms show that 91.9% of the comparisons for Rust and 98.5% for AS fall within the range of (97.5%, 100%). Moreover, 81.42% of the Rust and 87.65% of the AS comparisons are exactly 100%.

Table 2. Causes for similarities ≤ 90% in (CCFG,SCFG) comparisons: In Rust, 4% (4958 of 123374) of the total (CCFG,SCFG) comparisons are ≤ 90%. In AS, 0.61% (5 of 810) of the total (CCFG, SCFG) comparisons are ≤ 90%

Lang	MF	GD	ME	ISL	PI	Compares
AS	20%	80%	-	-	-	0.61%
Rust	2.16%	-	8.21%	89.59%	0.04%	4%

The results show that most SCFGs have a high degree of structural similarity to their corresponding CCFGs, with the vast majority being either identical or nearly identical. Although only a small percentage of the comparisons fall below 100%, we investigate the causes for the lower similarities. This task, however, requires a manual analysis of each individual comparison. So, to limit our sample, we manually analysed all the cases with similarity ≤ 90%, as this subset represents the lowest-performing comparisons and is sufficiently large to provide meaningful insights into the causes of lower similarity. We manually analysed 5 out of the total 810 comparisons for AS, and 4958 out of the total 123374 comparisons for Rust. Our analysis revealed that a lower similarity is due to one of the following causes, summarised in Table 2:

MF The high-level function compiles to *multiple functions* (MF). Each Wasm function represents a part of the high-level function. So, each WCFG and its corresponding SCFG represent only a portion of the high-level function. Instead, the CCFG represents the entire high-level function, thus resulting in lower similarity values, as the SCFG can only correspond to a part of the CCFG.

ISL The compiler generates *invalid source locations* (ISL) in the source map. This means that the Wasm address maps to an incorrect source location, as it may point to Rust comments, a pre-processor directive, an nonexistent line or column number, and so on. This is an issue that affects DWARF-based debuggers, including ours, causing what is known as *inconsistent debugging* behaviour [19]. In our case, this affects the structure of the SCFGs.

GD *Granularity difference* (GD) between an SCFG and CCFG negatively impacts the similarity value. For instance, a boolean expression may become a single node in a CCFG and an if-structure in an SCFG, as the evaluation of the second argument depends on the first.

PI The Rust compiler inserts *panic instructions* (PI) in the Wasm function. When executed, these instructions halt the runtime to prevent further execution with an incorrect state [41]. Panic instructions become nodes in the SCFGs, but the compiler does not include them in the CCFGs, thus affecting the similarity value.

ME In Rust, functions contain a *macro that expands* (ME) before the creation of the CCFGs, but the compiler does not generate sourcemappings for the expanded macro. As a result, the CCFG contains additional nodes and edges for the macro expansion, whereas the SCFG only has nodes and edges for the macro statement.

All comparisons ≤ 90 that we manually analysed include all the execution paths, except for the *ISL* cases, which may miss nodes or edges due to faulty debugging information. While SCFGs may include approximations (e.g., extra edges), this is not a concern as long as the SCFG is complete. The extra edges can cause the debugger to set both valid and unnecessary breakpoints, as the latter are not hit at runtime. Based on the results of this benchmark, we can confidently assert that the SCFGs capture all the execution paths when debugging information is present and correct. For most comparisons, the SCFGs are structurally similar to the CCFGs.

5.2.2 Experiment 2: Examining the Call Nodes.

Algorithm 2 constructs *call nodes* that hold references to all the functions they could call. This approach is by design conservative to ensure call node completeness. To validate this, we ran two benchmarks from Lehmann et al. [48] that assess call node completeness using a set of 24 microbenchmarks and 10 real-world binaries. Due to the page limitation, we only present the benchmark results. The benchmark details are provided in Appendix C.

The benchmark results confirm that our approach yields complete call nodes, succeeding on all 24 microbenchmarks and 9 out of 10 real-world binaries. We could not evaluate on one real-world binary due to the benchmark crashing during setup.

5.3 RQ3: Measuring Debugging Operations Performance

To assess the performance of our approach, we conduct an experiment that compares CFG-based debugging operations to those provided by a state-of-the-art MCU VM debugger. In particular, we compare the execution speed of our prototype's debugging operations to ones of WARDuino [82]. WARDuino's debugger implements operations in a *bytecode stepwise* manner, i.e., continuously stepping the VM to the next bytecode until the desired Wasm address is reached. We call this type of debugger *Stepwise* in the remainder. In contrast, our CFG-based debugger uses breakpoints to advance the computation to desired source code locations.

For this experiment, we deploy a factorial Rust implementation (code in Appendix D) on an M5StickC board [75] and execute it using WARDuino. The implementation contains a *main* that continuously calls the factorial with argument 5. Then we apply debugging operations upon different source locations and measure per operation, the execution time across 80 runs while excluding the first 30 due to warm-up. The applied operations are: step-into (si) and step-over (so) on the factorial called by the *main*, a *si* on the branching condition of the *if* statement defined in the factorial, and a step-out (*out*) after entering the body of the factorial.

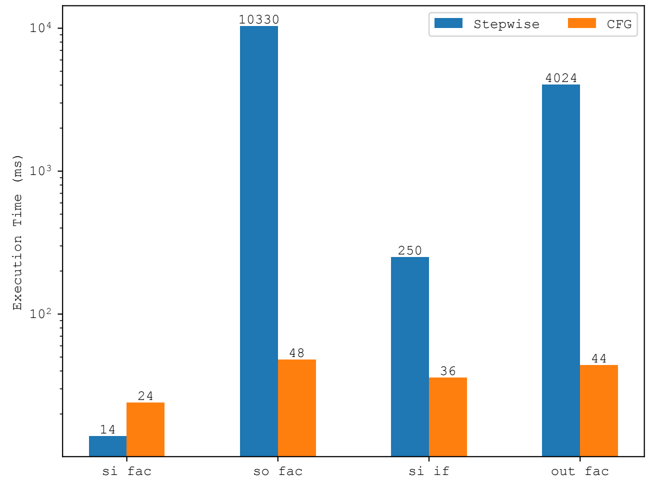
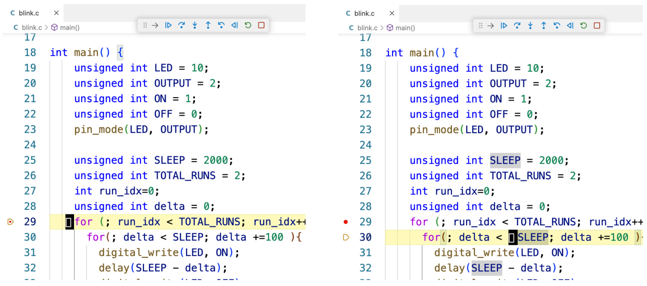


Figure 7. The mean execution times for Stepwise and CFG step-into (si), step-over (so), and step-out (out), measured across 80 runs (30 warm-up). We apply the operations to different source locations. The time is on a logarithmic scale. Each bar displays the mean execution time expressed in ms.

Using the obtained time measurements, we computed the mean execution times per operation (Figure 7). Detailed metrics can be found in Appendix D. The bar chart uses a logarithmic scale and reveals that CFG debugging outperforms Stepwise. While CFG *si if* is 7 times faster and CFG *out* is 91 times faster, these differences do not drastically impact the debugging experience. In contrast, the CFG *so fac* is 215 times faster, reducing the developer's average wait from 10 secs to just 48 ms, significantly improving the debugging experience. The *si fac* is the only case where CFG is slightly slower (1.74x), which highlights the overhead of CFG operations (i.e., find the start node and destination nodes, setting and removing breakpoints), only noticeable when executing a small number of instructions, as in this case.

In conclusion, CFG-based debugging operations outperform WARDuino's Stepwise ones, particularly when applying debugging operations that involve numerous instructions, such as step-over and step-out. When a relatively low number of instructions is executed, then CFG is slower than Stepwise, but the difference is acceptable.

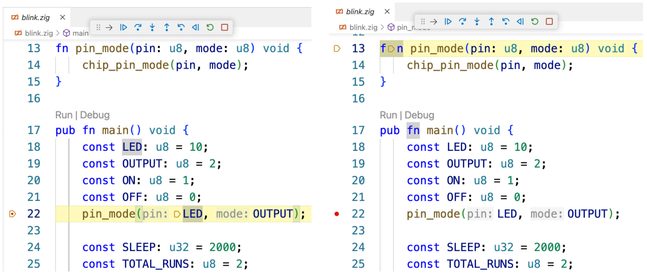
Construction Time of SCFGs. In this experiment, we did not factor in the construction time of SCFGs. The construction of SCFGs occurs during the debugger startup and takes approximately 114 ms to complete for the factorial: 105 ms parsing the Wasm module and 9 ms for building the WCFGs and SCFGs. This time is negligible compared to the time



```
17
18  int main() {
19      unsigned int LED = 10;
20      unsigned int OUTPUT = 2;
21      unsigned int ON = 1;
22      unsigned int OFF = 0;
23      pin_mode(LED, OUTPUT);
24
25      unsigned int SLEEP = 2000;
26      unsigned int TOTAL_RUNS = 2;
27      int run_idx=0;
28      unsigned int delta = 0;
29      for (; run_idx < TOTAL_RUNS; run_idx+
30        for(; delta < SLEEP; delta +=100 ){
31          digital_write(LED, ON);
32          delay(SLEEP - delta);
```

```
17
18  int main() {
19      unsigned int LED = 10;
20      unsigned int OUTPUT = 2;
21      unsigned int ON = 1;
22      unsigned int OFF = 0;
23      pin_mode(LED, OUTPUT);
24
25      unsigned int SLEEP = 2000;
26      unsigned int TOTAL_RUNS = 2;
27      int run_idx=0;
28      unsigned int delta = 0;
29      for (; run_idx < TOTAL_RUNS; run_idx++
30        for(; delta < SLEEP; delta +=100 )
31          digital_write(LED, ON);
32          delay(SLEEP - delta);
```

Figure 8. The VSCode debugger targeting the C blink application. The figures highlight a step operation that advances computation from line 29 (left) to line 30 (right).



```
13  fn pin_mode(pin: u8, mode: u8) void {
14      chip_pin_mode(pin, mode);
15  }
16

17  pub fn main() void {
18      const LED: u8 = 10;
19      const OUTPUT: u8 = 2;
20      const ON: u8 = 1;
21      const OFF: u8 = 0;
22      pin_mode(pin: LED, mode: OUTPUT);
23
24      const SLEEP: u32 = 2000;
25      const TOTAL_RUNS: u8 = 2;
```

```
13  fn pin_mode(pin: u8, mode: u8) void {
14      chip_pin_mode(pin, mode);
15  }
16

17  pub fn main() void {
18      const LED: u8 = 10;
19      const OUTPUT: u8 = 2;
20      const ON: u8 = 1;
21      const OFF: u8 = 0;
22      pin_mode(pin: LED, mode: OUTPUT);
23
24      const SLEEP: u32 = 2000;
25      const TOTAL_RUNS: u8 = 2;
```

Figure 9. The VSCode debugger targets the Zig blink application. The figures highlight reaching a breakpoint at line 22 (left), followed by a step into `pin_mode` (right).

required for flashing the Wasm module and VM into the MCU, or just the module if the VM is already present. So, the SCFGs' construction time does not affect the findings of this benchmark since an initial slower startup in the prototype is compensated for once debugging operations begin.

5.4 RQ4: Extending Debugging Support to C and Zig

Finally, we assess whether it is possible to extend CFG debugging to various languages that compile to WebAssembly. In the previous sections, we showed that our CFG-based debugger prototype supports Rust and AssemblyScript. We now show that our debugger also supports applications written in Zig and C [14, 44]. In doing so, we show that our prototype works out-of-the-box for languages that compile to Wasm as long as their compilers: (1) generate Wasm bytecode version 1 and (2) produce DWARF or Source Map Spec. This is because a Wasm module and debugging information are the only inputs our algorithm needs to build SCFGs, which in turn enable CFG-based debugging operations.

In this experiment, we use the C blink application introduced in Section 2.1, along with its equivalent Zig implementation, shown in Appendix A.2. Using their respective compilers, clang for C [14] and the Zig compiler for Zig [44], we compile these applications to Wasm version 1, along with their associated DWARF debugging information. With these outputs, our prototype builds SCFGs for each blink application. Using a simple debug scenario, we show how our CFG VSCode debugger prototype successfully targets each blink implementation by using the generated SCFGs. For the C version, after starting the debugger, we set a breakpoint at line 29 and resume the execution. The application pauses at the set breakpoint (left Figure 8). Then, we apply a *step*, which causes the debugger to access the SCFG node corresponding to the paused source location, i.e., node *s*2 in Figure 3b. Using *s*2, the debugger identifies the nodes to where execution should advance (i.e., *s*3, *s*4), sets breakpoints on those nodes, and resumes execution. The application pauses upon reaching *s*4, i.e., line 30 (right Figure 8). In the Zig version, we set a breakpoint at line 22 and resume execution. Upon reaching the breakpoint (left Figure 9), we step into the `pin_mode` function (right Figure 9).

Both debug scenarios show that the prototype can target both languages. Extending to other languages [16, 81] should, in theory, not require any modification to the debugger, as these languages also compile to Wasm version 1 and generate DWARF or Source Map Spec. Moreover, the C debug scenario shows how CFG debugging can reduce inconsistencies with the step command. Concretely, the debugger correctly steps from line 29 to 30 when encountering a double for loop without initialisers, unlike GDB/LLDB's step command, which, as reported by Yan et al. [98], bypasses line 30.

6 Related Work

In the literature, several language-agnostic debuggers have been proposed. Debuggers, such as GDB [29] and LLDB [50], rely on DWARF [20] to offer language-agnostic debugging. However, the language-agnostic debugging support is only for languages that compile to native desktop code and produce DWARF, such as Rust, Go, and Zig. Similarly, GDB forks [23, 24] for MCUs support only native code and offer no source-level debugging of bytecode applications running on MCU VMs. Enabling this requires implementing a debug server and tedious low-level logic to map CPU to bytecode state using OpenOCD [66]. Our approach avoids the need for such low-level logic as it does not depend on OpenOCD for debugging. Moreover, these debuggers suffer from *inconsistent debugging behaviour* when targeting unoptimised native code, such as skipping reachable source locations, as shown by Yang et al. [98]. In contrast, CFG debugging helps mitigate such issues, as outlined in Section 2.2 and demonstrated in Section 5.4. However, our approach can also suffer from inconsistent debugging behaviour when the compiler produces incorrect debugging information.

In the context of managed runtimes, Truffle/GraalVM offers language-agnostic debugging for the languages built on the Truffle framework [86]. Unfortunately, this approach is too memory and computational-intensive to apply to MCUs.

WebAssembly VMs for desktops or browsers offer some support for language-agnostic debugging. Wasmtime [10] incorporates a debugger able to debug Wasm programs, but only if they include DWARF debugging information. Thus, working for languages like C/C++, Rust, which can emit DWARF. However, unlike our approach, it cannot debug AssemblyScript programs. Moreover, Wasmtime relies on just-in-time (JIT) compilation of Wasm to native code to enable debugging. However, the JIT infrastructure is too memory-intensive to run on MCUs and is architecture-dependent [96]. The latter is particularly problematic in MCUs due to the *combinatorial explosion problem* [54], which states that new MCU hardware vendors are continuously emerging. Chrome also provides a debugger [30] for Wasm-compatible languages that generate DWARF, but similarly, due to the hardware constraints, cannot be ported to MCUs. There exists debugging support for JavaScript combined with Wasm, but such debuggers do not extend to new languages [42].

Some Wasm VMs targeted for MCUs, such as WAMR [2] and Wasm3 [46], only offer debugging on desktops, not on MCUs. To our knowledge, WARDuino [36] is the only VM offering a Wasm debugger on MCUs, supporting AssemblyScript and partially supporting Rust [82]. However, its debugging operations can be slow due to round-trips between the debugger front-end and the VM running on the MCU, as shown in our evaluation Section 5.3.

There are several managed languages running on MCU VMs [8, 28, 32, 62], offering debugging for a specific language. While MicroPython [28] offers only debugging support on super-sensors like the Raspberry Pi [64], both Espruino and Duktape [32, 62] offer remote debugging support on MCUs.

Finally, CFGs are typically built from source code [1, 6, 14, 41, 51] or assembly code [45, 63, 87] and can be generated at three levels: source level (SL), intermediate level (IR), or binary level (BL). Specific to Wasm, existing approaches construct BL [55, 76, 92] or IR [91] CFGs starting from a Wasm module or source code. However, these solutions are either compiler-dependent or produce CFGs lacking a clear link between source code and bytecode. While CFGs are widely used in static analysis tasks such as malware detection [38, 43, 53, 57, 103], compiler optimisations (e.g., liveness analysis, dead code elimination) [1], software rewriting and generation [9, 56, 90], test coverage evaluation [65], static bug detection [102], and aiding in bug reproducibility [100]. Notably, CFGs have been used to address inconsistencies that arise with the source-level state when debugging optimised native code [95] by using the graphs to delay the execution of machine instructions. In contrast, our work uses CFGs to enable stepping commands. To the best of our knowledge, our work is the first to employ CFGs for debugging Wasm programs.

7 Discussion and Future Work

In this section, we discuss in more detail several limitations of our approach, as well as future work pointers.

Debugging Optimised Code. Compiler optimisations impact the availability of debugging information, which in turn affects CFG completeness and debugging correctness. However, even with the optimisations, SCFGs can still be constructed from the available source mappings. In practice, as with GDB/LLDB, when stepping through optimised code with our CFG debugger, locations that miss a mapping will not be reached. A more thorough evaluation is needed to draw robust conclusions about which optimisations CFG debugging can support.

Debugging on General Platforms. Debugging of Wasm is supported outside MCUs (e.g., in browsers [31] or desktops [10]). Although it was not the focus of this paper, we believe that our approach could be useful to such Wasm debuggers. First, it reduces inconsistent stepping as discussed in Section 5.4. Second, it enables the debugging of languages that do not produce DWARF (e.g., AssemblyScript), as only a sourcemap is needed in the debugging information. Third, it enables debugging operations that DWARF-based debuggers cannot support. For instance, *step to the next loop iteration* requires knowledge of control flow, absent in DWARF.

Debugging Hardware Errors. With our approach, we cannot debug MCU properties such as hardware errors or erratic behaviour due to low battery life [60]. To address this, CFG debugging is best complemented with runtime debugging information, possibly enabled through a more advanced debugging API. For instance, exposing battery status allows the debugger to adjust features based on available power.

8 Conclusion

We introduced a debugging technique based on CFGs specialised for debugging. Our approach is language-agnostic for three reasons. First, the debugger can be implemented in any language to integrate with various IDEs (e.g., VSCode, IntelliJ). Second, the debugger targets any Wasm language, provided the compiler emits debugging information containing a *sourcemap*. This is already the case for many Wasm compilers, including Rust [41], emscripten (C/C++) [14], AssemblyScript [6], and TinyGo [81]. Third, debugging operations are implemented based on the graph structure, rather than the semantics or syntax of the source code language.

We implemented a CFG debugger and integrated it into the WARDuino Wasm VM using four basic operations of the VM's debug API. Our prototype enables (1) classic stepping operations (e.g., step into, step over) and (2) debugging of four Wasm languages. Our evaluation showed that the prototype outperforms WARDuino's debugger and has the potential to debug any language that compiles to Wasm and generates debugging information, without requiring modifications to the prototype's software.

Acknowledgments

We are deeply grateful for the valuable feedback provided by the anonymous reviewers and for the insightful suggestions from our shepherds, Stephen Kell and David Leopoldseder. This research is funded by the Fonds Wetenschappelijk Onderzoek - Vlaanderen (FWO) SB fellowship 1SHEU24N.

A Blinking LED Applications

A.1 C Implementation

The complete blinking LED application as introduced in Section 2.1. The application compiles to Wasm for deployment on WARDuino. To compile to Wasm, we ran the following command using clang version 20.1.8: *clang blink.c –target=wasm32 -nostdlib -Wl,–no-entry,–export-all,–allow-undefined -g -o blink.wasm*

```c
extern void chip_pin_mode(unsigned int, unsigned int);
extern void chip_digital_write(unsigned int, unsigned
     int);
extern void chip_delay(unsigned int);


void pin_mode(unsigned int pin, unsigned int mode) {
    chip_pin_mode(pin, mode);
}

void digital_write(unsigned int pin, unsigned int state)
      {
    chip_digital_write(pin, state);
}

void delay(unsigned int ms) {
    chip_delay(ms);
}

int main() {
    unsigned int LED = 10;
    unsigned int OUTPUT = 2;
    unsigned int ON = 1;
    unsigned int OFF = 0;
    pin_mode(LED, OUTPUT);

    unsigned int SLEEP = 2000;
    unsigned int TOTAL_RUNS = 2;
    int run_idx=0;
    unsigned int delta = 0;
    for (; run_idx < TOTAL_RUNS; run_idx++){
      for(; delta < SLEEP; delta +=100 ){
        digital_write(LED, ON);
        delay(SLEEP - delta);
        digital_write(LED, OFF);
        delay(SLEEP - delta);
      }
      delta = 0;
    }
}
```

A.2 Zig Implementation

The following listing is equivalent to Appendix A.1, written in Zig and compiled to Wasm for deployment on WARDuino. To compile to Wasm, we ran the following command using zig version 0.14.1: *zig cc -z stack-size=32768 -target wasm32-freestanding blink.zig -o blink.wasm*

```zig
extern fn chip_pin_mode(u32, u8) void;
extern fn chip_digital_write(u8, u8) void;
extern fn chip_delay(u32) void;

fn digital_write(pin: u8, value: u8) void {
    chip_digital_write(pin, value);
}

fn delay(ms: u32) void {
    chip_delay(ms);
}

fn pin_mode(pin: u8, mode: u8) void {
    chip_pin_mode(pin, mode);
}

pub fn main() void {
    const LED: u8 = 10;
    const OUTPUT: u8 = 2;
    const ON: u8 = 1;
    const OFF: u8 = 0;
    pin_mode(LED, OUTPUT);

    const SLEEP: u32 = 2000;
    const TOTAL_RUNS: u8 = 2;
    var delta: u32 = 0;

    for (0..TOTAL_RUNS) |_| {
        for (0..SLEEP) |_| {
            delta += 100;
            if (delta > SLEEP) break;
            digital_write(LED, ON);
            chip_delay(SLEEP - delta);
            digital_write(LED, OFF);
            chip_delay(SLEEP - delta);
        }
        delta = 0;
    }
}
```

B Debugging Operations

This section illustrates the listings of *step over* and *step out* as introduced in Section 4.2.

Step Over. The following shows a JavaScript-like pseudocode of a CFG step over.

```
function stepOver(scfgs,sn){
 const ns = neighbours(sn)
 if (ns.length !== 0)
    return ns
5  return stepOut(scfgs, sn)
6 }
```

Using the start node sn, step over accesses all the outgoing neighbours of sn (line 2). If at least one neighbour is found, those are returned as destination nodes (lines 3-4). Otherwise, step over performs a step out.

Step Out. The following shows a JavaScript-like pseudo-code of a CFG step out.

```
1  function stepOut(scfgs,sn){
2   const scfg = getSCFG(sn)
3   const f = funID(scfg)
4   const cn = callNodes(scfgs,f)
5   const dns = []
6   for (const c of cn){
7    const ns = stepOver(scfgs,c)
8    dns.push(...ns)
9   }
10  return dns;
11 }
```

Using the start node, step out first accesses the SCFG associated with the start node (line 2). This is then used to access the function identifier f for which the SCFG was created (line 3). Then the operation accesses all the call nodes that call f using the helper function `callNodes` (line 4). Per call node, the operation applies a step over and collects all the obtained nodes as destination nodes (lines 6-8). Finally, step out returns the collected destination nodes (line 10).

```
#![no_std]
#![no_main]
use core::panic::PanicInfo;

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}

fn fac(n: u64) -> u64 {
    if n == 0 || n == 1 {
        1
    } else {
        n * fac(n - 1)
    }
}

#[no_mangle]
pub fn main() {
    loop {
        fac(5);
    }
}
```

Figure 10. The Rust factorial used in Section 5.3.

C Experiment Call Nodes Completeness

In this section, we detail the experiment that we conducted to evaluate call node completeness. Specifically, to assess whether the call nodes refer to all the functions they could call. We build on the idea that the call nodes of a function f represent a call and callee relationship between f and all the functions called by f. If we express this relationship as a graph and do the same for the functions called by f, we obtain the *call graph* of f. As defined by Lehmann et al. [48], if the obtained call graph misses no node or edge, we say that the call graph is *sound*. Therefore, to determine if all call nodes refer to all the functions they could call, we construct a call graph from all the call nodes starting from the main of the program. If we show that the obtained call graph is sound, we can conclude that the call nodes refer to all the functions they could call since they have been used to build the call graph.

For this experiment, we use the *wasm-call-graphs* benchmarks proposed by Lehman et al. [48, 72]. These benchmarks consist of 24 *microbenchmarks* and 10 *real-world binaries* that measure call graph soundness.

Microbenchmarks. The microbenchmarks are designed to evaluate soundness across scenarios such as the use of *implicit entry functions*, *functions exported through tables*, *indirect calls*, and so on. For each microbenchmark, we construct a call graph from the call nodes. To determine call graph soundness, the benchmark compares our call graph to a ground truth, i.e., a sound call graph pre-constructed by the authors [48]. The results of the benchmark are illustrated in Table 3. From the results, we observe that our call graphs are sound across all 24 examples.

Real-world Binaries. The real-world binaries benchmark comprises Wasm modules compiled from 10 libraries such as SQL.js [74] (a Wasm port of SQLite), Graphviz [33] (a C++ graph visualisation library), and more. To measure soundness, the authors define three values per Wasm module:

1. $|\mathrm{F}_{all}|$ the number of Wasm functions.
2. $|\mathrm{F}_{r}|$ the number of reachable Wasm functions.
3. $|\mathrm{F}_{dyn}|$ is the number of Wasm functions that are dynamically called by test cases included with each library.

The authors [48] also define $F_{unsound} = F_{dyn} - F_r$, i.e., the set of functions that cannot be removed. A value greater than zero for $|F_{unsound}|$ indicates that the call graph is unsound.

Using these real-world binaries, the benchmark builds call graphs using our approach and computes $|F_{unsound}|$ and $|F_r|$. The results, shown in Table 4, demonstrate that our call graphs are not unsound for any of the libraries. No results are available for OpenCV as the benchmark crashed during the installation of the library.

Table 3. The results of applying the microbenchmarks to our approach: ✓stands for success.

Description	Sound
Simple direct call	✓
Transitive direct call	✓
Direct call to imported function	✓
Implicit entry point: Wasm start section	✓
Implicit entry point: Wasi start section	✓
Imported host code calls exported function	✓
Functions in exported table are reachable	✓
Functions in imported table are reachable	✓
Table is mutable by host	✓
Table init. offset is imported from host	✓
Memory init. offset is imported from host	✓
Functions must be in table for indirect call	✓
Types can constrain indirect call targets	✓

Description	Sound
Constant table index value	✓
Index value data-flow through local variable	✓
Masked index value	✓
Inter-procedural index value, parameter	✓
Inter-procedural index value, function result	✓
Index from memory, constant address	✓
Index from memory, address inter-procedural, parameter	✓
Index from memory, address inter-procedural, result	✓
Index from memory, double indirect load	✓
Index from memory, memory is mutable	✓
C++ virtual calls from unrelated classes	✓

Table 4. The $|F_{unsound}|$ measurement for our approach across the real-world binaries. Constructed call graphs are unsound when $|F_{unsound}|$ is greater than zero. OpenCV could not be benchmarked due to an installation error (indicated by -).

Library	$\|F_{all}\|$	$\|F_{dyn}\|$	$\|F_r\|$	$\|F_{unsound}\|$
opencv	10909	871	-	-
sql.js	1261	390	1261	0
rsa	785	274	779	0
blake	81	21	76	0
libmagic	736	213	736	0
graphviz	2018	790	2018	0
source-map	46	19	38	0
shiki	213	105	213	0
fonteditor	1118	381	1118	0
opusscript	356	169	356	0

Table 5. The execution times of step-into, step-over, and step-out, using the Stepwise (S) WARDuino debugger [82] and our CFG (C) prototype. The metrics are computed from 80 runs (30 warm-up). The operations are applied at different source locations (line *l*, column *c*) on the factorial example.

operation	DBG	min (ms)	mean (ms)	max (ms)	std. dev.
step-into fac	S	11.76	13.8	17.5	1.95
(line 21, col 9)	C	23.45	24.02	29.73	0.84
step-over fac	S	10171.94	10330.28	10667.94	119.29
(line 21, col 9)	C	47.8	48.14	50.99	0.43
step-into if condition	S	239.79	249.71	279.84	8.53
(line 11, col 8)	C	35.52	36.22	48.52	1.95
step-out fac	S	3875.85	4024.0	4284.05	81.51
(line 14, col 13)	C	40.24	44.09	64.61	3.24

D Debugging Operations Detailed Metrics

In this section, we provide the accompanying results of the benchmark executed in Section 5.3. In particular, Figure 10 shows the complete code used for the benchmark, and Table 5 presents the statistical metrics computed for each debugging operation. The factorial example was compiled using Rust version 1.81.0 with the following compile command: *rustc -C link-args=–no-entry -C link-args=-zstack-size=32768 –target wasm32-unknown-unknown -g fac.rs*

References

[1] Alfred Aho and Monlca S Lam. 2022. *Compilers principles techniques & tools*. Pearson.

[2] Bytecode Alliance. 2019. WebAssembly Micro Runtime (WAMR). https://github.com/bytecodealliance/wasm-micro-runtime. Accessed: August 24, 2025.

[3] Bytecode Alliance. 2024. CLI and Rust libraries for low-level manipulation of WebAssembly modules. https://github.com/bytecodealliance/wasm-tools. Accessed: August 24, 2025.

[4] Arduino. 2025. Arduino. https://www.arduino.cc/. Accessed: August 14, 2025.

[5] as2d. 2019. as2d. https://github.com/as2d/as2d/. Accessed: August 24, 2025.

[6] AssemblyScript. 2017. A TypeScript-like language for WebAssembly. https://www.assemblyscript.org/. Accessed: August 24, 2025.

[7] AssemblyScript. 2017. AssemblyScript. https://github.com/AssemblyScript/assemblyscript. Accessed: August 24, 2025.

[8] AtomVM. 2017. Tiny Erlang VM. https://github.com/atomvm/AtomVM. Accessed: August 24, 2025.

[9] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 257–269. doi:10.1145/2771783.2771796

[10] bytecodealliance. 2017. Wasmtime. https://docs.wasmtime.dev/. Accessed: August 24, 2025.

[11] Qing Cao, Tarek Abdelzaher, John Stankovic, Kamin Whitehouse, and Liqian Luo. 2008. Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems* (Raleigh, NC, USA) *(SenSys '08)*. Association for Computing Machinery, New York, NY, USA, 85–98. doi:10.1145/1460412.1460422

[12] casper ecosystem. 2020. cep18. https://github.com/casper-ecosystem/cep18. Accessed: August 24, 2025.

[13] Patrick P.F. Chan and Christian Collberg. 2014. A Method to Evaluate CFG Comparison Algorithms. In *2014 14th International Conference on Quality Software*. 95–104. doi:10.1109/QSIC.2014.28

[14] Clang. 2007. Clang: a C language family frontend for LLVM. https://clang.llvm.org/. Accessed: August 24, 2025.

[15] ColinEberhardt. 2017. wasm-mandelbrot. https://github.com/ColinEberhardt/wasm-mandelbrot/. Accessed: August 24, 2025.

[16] Glasgow Haskell Compiler. 2002. Welcome to the GHC User's Guide. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/. Accessed: August 24, 2025.

[17] Frank Denis. 2018. wasm-crypto. https://github.com/jedisct1/wasm-crypto. Accessed: August 24, 2025.

[18] DHTMLX. 2019. excel2json. https://github.com/DHTMLX/excel2json. Accessed: August 24, 2025.

[19] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. Who's debugging the debuggers? exposing debug information bugs in optimized binaries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1034–1045. doi:10.1145/3445814.3446695

[20] DWARF. 1989. DWARF Debugging Format. https://dwarfstd.org/. Accessed: August 24, 2025.

[21] Espessif. 2020. Introduction to ESP-Prog Board. https://docs.espressif.com/projects/esp-iot-solution/en/latest/hw-reference/ESP-Prog_guide.html. Accessed: August 24, 2025.

[22] ESPESSIF. 2025. Espressif. https://www.espressif.com/. Accessed: August 14, 2025.

[23] Esspresif. 2018. ESP32 JTAG Debugging Xtensa. https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/jtag-debugging/index.html. Accessed: August 24, 2025.

[24] Esspresif. 2021. ESP32C3 JTAG Debugging riscv32. https://docs.espressif.com/projects/esp-idf/en/v5.0.8/esp32c3/api-guides/jtag-debugging/index.html. Accessed: August 24, 2025.

[25] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 480–491. doi:10.1145/2976749.2978370

[26] FreeRTOS. 2025. FreeRTOS. https://www.freertos.org/. Accessed: August 14, 2025.

[27] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 896–899. doi:10.1145/3238147.3240480

[28] Damien George. 2013. MicroPython. https://micropython.org/. Accessed: August 24, 2025.

[29] GNU. 1986. The GNU Project Debugger. https://www.gnu.org/software/gdb/. Accessed: August 24, 2025.

[30] Google. 2009. Chrome. https://www.google.com/chrome/. Accessed: August 24, 2025.

[31] Google. 2024. Debug C/C++ WebAssembly. https://developer.chrome.com/docs/devtools/wasm. Accessed: Dec 24, 2024.

[32] Gordon Williams. 2012. Espruino. https://www.espruino.com/. Accessed: August 24, 2025.

[33] Graphviz. 1991. Graphviz. https://graphviz.org/. Accessed: August 24, 2025.

[34] Robert Griesemer, Rob Pike, and Ken Thompson. 2009. Go. https://go.dev/. Accessed: August 24, 2025.

[35] IEEE 1149.1 Working Group. 2009. Official IEEE Std. 1149.1 Standard Working Group. https://grouper.ieee.org/groups/1149/1/. Accessed: August 24, 2025.

[36] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes - MPLR 2019*. ACM Press, 27–36. doi:10.1145/3357390.3361029

[37] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. doi:10.1145/3062341.3062363

[38] Nguyen Minh Hai, Mizuhito Ogawa, and Quan Thanh Tho. 2016. Obfuscation Code Localization Based on CFG Generation of Malware. In

Foundations and Practice of Security, Joaquin Garcia-Alfaro, Evangelos Kranakis, and Guillaume Bonfante (Eds.). Springer International Publishing, Cham, 229–247.

[39] Prasanna Hambarde, Rachit Varma, and Shivani Jha. 2014. The Survey of Real Time Operating System: RTOS. In *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*. 34–39. doi:10.1109/ICESC.2014.15

[40] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Comput. Surv.* 54, 3, Article 51 (April 2021), 38 pages. doi:10.1145/3446371

[41] Graydon Hoare. 2012. Rust. https://www.rust-lang.org/. Accessed: August 24, 2025.

[42] Philémon Houdaille, Djamel Eddine Khelladi, Benoit Combemale, Gunter Mussbacher, and Tijs van Der Storm. 2025. PolyDebug: A Framework for Polyglot Debugging. *The Art, Science, and Engineering of Programming* 10, 1 (2025). doi:10.22152/programming-journal.org/2025/10/13

[43] Arun Kanuparthi, Jeyavijayan Rajendran, and Ramesh Karri. 2016. Controlling your control flow graph. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 43–48. doi:10.1109/HST.2016.7495554

[44] Andrew Kelley. 2016. Zig. https://ziglang.org/. Accessed: August 24, 2025.

[45] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021 (28th Annual Network and Distributed System Security Symposium, NDSS 2021)*. The Internet Society. doi:10.14722/ndss.2021.24386 Publisher Copyright: © 2021 28th Annual Network and Distributed System Security Symposium, NDSS 2021. All Rights Reserved.; 28th Annual Network and Distributed System Security Symposium, NDSS 2021 ; Conference date: 21-02-2021 Through 25-02-2021.

[46] Wasm3 Labs. 2019. wasm3. https://github.com/wasm3/wasm3. Accessed: August 24, 2025.

[47] Tom Lauwaerts, Carlos Rojas Castillo, Robbert Gurdeep Singh, Matteo Marra, Christophe Scholliers, and Elisa Gonzalez Boix. 2022. Event-Based Out-of-Place Debugging. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 85–97. doi:10.1145/3546918.3546920

[48] Daniel Lehmann, Michelle Thalakottur, Frank Tip, and Michael Pradel. 2023. That's a Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 892–903. doi:10.1145/3597926.3598104

[49] Chao Li, Rui Chen, Boxiang Wang, Zhixuan Wang, Tingting Yu, Yunsong Jiang, Bin Gu, and Mengfei Yang. 2023. An Empirical Study on Concurrency Bugs in Interrupt-Driven Embedded Software. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1345–1356. doi:10.1145/3597926.3598140

[50] LLVM. [n. d.]. The LLDB Debugger. https://lldb.llvm.org/. Accessed: August 24, 2025.

[51] LLVM. 2003. The LLVM Compiler Infrastructure. https://llvm.org/. Accessed: August 24, 2025.

[52] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177. doi:10.1109/TSE.2017.2655046

[53] Zhuo Ma, Haoran Ge, Yang Liu, Meng Zhao, and Jianfeng Ma. 2019. A Combination Method for Android Malware Detection Based on Control Flow Graphs and Machine Learning Algorithms. *IEEE Access* 7 (2019), 21235–21245. doi:10.1109/ACCESS.2019.2896003

[54] Amir Makhshari and Ali Mesbah. 2021. IoT Bugs and Development Challenges. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 460–472. doi:10.1109/ICSE43902.2021.00051

[55] Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão. 2022. Concolic Execution for WebAssembly. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:29. doi:10.4230/LIPIcs.ECOOP.2022.11

[56] Xiaozhu Meng and Weijie Liu. 2021. Incremental CFG patching for binary rewriting. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1020–1033. doi:10.1145/3445814.3446765

[57] Shaswata Mitra, Stephen A. Torri, and Sudip Mittal. 2023. Survey of Malware Analysis through Control Flow Graph using Machine Learning. In *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 1554–1561. doi:10.1109/TrustCom60117.2023.00212

[58] Mozilla. 2004. Firefox. https://www.mozilla.org/. Accessed: August 24, 2025.

[59] Mozilla. 2011. source-map. https://github.com/mozilla/source-map. Accessed: August 24, 2025.

[60] Murat Mülayim, Arda Goknil, and Kasım Sinan Yıldırım. 2020. Taskify: An Integrated Development Environment to Develop and Debug Intermittent Software for the Batteryless Internet of Things. In *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*. 348–355. doi:10.1109/DCOSS49796.2020.00062

[61] Opera. 1995. Opera. https://www.opera.com/. Accessed: August 24, 2025.

[62] Duktape organization. 2014. Duktape. https://duktape.org/. Accessed: August 24, 2025.

[63] Jannik Pewny and Thorsten Holz. 2013. Control-flow restrictor: compiler-based CFI for iOS. In *Proceedings of the 29th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '13)*. Association for Computing Machinery, New York, NY, USA, 309–318. doi:10.1145/2523649.2523674

[64] Raspberry Pi. 2012. Raspberry Pi Foundation. https://www.raspberrypi.org/. Accessed: August 24, 2025.

[65] Ani Rahmani, Joe Lian Min, and Asri Maspupah. 2020. An Evaluation of Code Coverage Adequacy in Automatic Testing using Control Flow Graph Visualization. In *2020 IEEE 10th Symposium on Computer Applications & Industrial Electronics (ISCAIE)*. 239–244. doi:10.1109/ISCAIE47305.2020.9108838

[66] Dominic Rath. 2005. Open On-Chip Debugger. https://openocd.org/. Accessed: August 24, 2025.

[67] Carlos Rojas Castillo, Matteo Marra, Jim Bauwens, and Elisa Gonzalez Boix. 2022. Out-of-things debugging: A live debugging approach for Internet of Things. *arXiv preprint arXiv:2211.01679* (2022).

[68] Sven Sauleau. 2024. webassemblyjs. https://github.com/xtuc/webassemblyjs. Accessed: August 24, 2025.

[69] Kevin Sendjaja, Satrio Adi Rukmono, and Riza Satria Perdana. 2021. Evaluating Control-Flow Graph Similarity for Grading Programming Exercises. In *2021 International Conference on Data and Software Engineering (ICoDSE)*. 1–6. doi:10.1109/ICoDSE53690.2021.9648464

[70] Silvia O'Dwyer. 2019. photon. https://github.com/silvia-odwyer/photon. Accessed: August 24, 2025.

[71] Oleg Sokolsky, Sampath Kannan, and Insup Lee. 2006. Simulation-Based Graph Similarity. In *Tools and Algorithms for the Construction and Analysis of Systems*, Holger Hermanns and Jens Palsberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 426–440.

[72] SOLA. 2022. wasm-call-graphs. https://github.com/sola-st/wasm-call-graphs. Accessed: August 24, 2025.

[73] SourceMap. 2025. SourceMap. https://tc39.es/source-map/. Accessed: August 24, 2025.

[74] SQL.JS. 2012. sql.js. https://github.com/sql-js/sql.js. Accessed: August 24, 2025.

[75] M5 Stack. 2025. M5 Stack. https://m5stack.com/. Accessed: August 24, 2025.

[76] Quentin Stiévenart and Coen De Roover. 2021. Wassail: a WebAssembly static analysis library. In *Fifth International Workshop on Programming Technology for the Future Web.*

[77] Quentin Stiévenart and Coen De Roover. 2020. Compositional Information Flow Analysis for WebAssembly Programs. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 13–24. doi:10.1109/SCAM51674.2020.00007

[78] Sven-Hendrik Haase. 2011. genact. https://github.com/svenstaro/genact. Accessed: August 24, 2025.

[79] Matthew Tancreti, Vinaitheerthan Sundaram, Saurabh Bagchi, and Patrick Eugster. 2015. TARDIS: software-only system-level record and replay in wireless sensor networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks* (Seattle, Washington) *(IPSN '15)*. Association for Computing Machinery, New York, NY, USA, 286–297. doi:10.1145/2737095.2737096

[80] The Rust Programming Language. 2010. rust. https://github.com/rust-lang/rust. Accessed: August 24, 2025.

[81] TinyGo. 2018. A Go Compiler For Small Places. https://tinygo.org/. Accessed: August 24, 2025.

[82] TOPLLab. 2022. WARDuino VSCode. https://github.com/TOPLLab/WARDuino-VSCode. Accessed: August 24, 2025.

[83] Aaron Turner. 2018. wasmboy. https://github.com/torch2424/wasmboy. Accessed: August 24, 2025.

[84] Aaron Turner. 2019. as-bind. https://github.com/torch2424/as-bind. Accessed: August 24, 2025.

[85] Turso Database. 2023. limbo. https://github.com/tursodatabase/limbo. Accessed: August 24, 2025.

[86] Michael Van De Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *The Art, Science, and Engineering of Programming* 2, 3 (March 2018). doi:10.22152/programming-journal.org/2018/2/14

[87] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *2016 IEEE Symposium on Security and Privacy (SP)*. 934–953. doi:10.1109/SP.2016.60

[88] Nischay Venkatram. 2020. as-benchmarks. https://github.com/nischayv/as-benchmarks. Accessed: August 24, 2025.

[89] Tao Wang, Kangkang Zhang, Wei Chen, Wensheng Dou, Jiaxin Zhu, Jun Wei, and Tao Huang. 2022. Understanding device integration bugs in smart home system. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 429–441. doi:10.1145/3533767.3534365

[90] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '12)*. Association for Computing Machinery, New York, NY, USA, 299–308. doi:10.1145/2420950.2420995

[91] WAVM. 2015. WAVM. https://wavm.github.io/. Accessed: August 24, 2025.

[92] WebAssembly. 2015. Binaryen. https://github.com/WebAssembly/binaryen. Accessed: August 24, 2025.

[93] WebAssembly. 2017. WebAssembly Spec. https://github.com/WebAssembly/spec. Accessed: August 24, 2025.

[94] WebAssembly. 2019. WebAssembly Specification. https://webassembly.github.io/spec/. Accessed: August 24, 2025.

[95] Le-Chun Wu, Rajiv Mirani, Harish Patil, Bruce Olsen, and Wen-mei W. Hwu. 1999. A new framework for debugging globally optimized code. *SIGPLAN Not.* 34, 5 (May 1999), 181–191. doi:10.1145/301631.301663

[96] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 136 (Oct. 2021), 30 pages. doi:10.1145/3485513

[97] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. 2007. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems* (Sydney, Australia) *(SenSys '07)*. Association for Computing Machinery, New York, NY, USA, 189–203. doi:10.1145/1322263.1322282

[98] Yibiao Yang, Maolin Sun, Jiangchang Wu, Qingyang Li, and Yuming Zhou. 2025. Debugger Toolchain Validation via Cross-Level Debugging. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 280–294. doi:10.1145/3669940.3707271

[99] yar2001. 2023. JavaScript-CFG. https://github.com/yar2001/JavaScript-CFG. Accessed: August 24, 2025.

[100] Cristian Zamfir and George Candea. 2010. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 321–334. doi:10.1145/1755913.1755946

[101] Zephyr Project. 2025. Zephyr. https://docs.zephyrproject.org/latest/. Accessed: August 14, 2025.

[102] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, Chunming Hu, and Yang Liu. 2023. Detecting Condition-Related Bugs with Control Flow Graph Neural Network. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1370–1382. doi:10.1145/3597926.3598142

[103] Zongqu Zhao. 2011. A virus detection scheme based on features of Control Flow Graph. In *2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC)*. 943–947. doi:10.1109/AIMSEC.2011.6010676