# 1 Introduction

As part of our research into software error mitigation, we recognized the necessity of checking for control flow errors along with dataflow errors.

# 2 Previous Work

Control-flow checking by software signatures

# 3 Algorithm

The algorithm we determined to use is one found in the research paper mentioned above. A brief description will be included here.

A program may be split into a representation using "basic blocks." A basic block ($b_n$) is a collection of sequential instructions, into which there is only one entry point, and out of which there is only one exit point. Many basic blocks may branch into a single basic block, and a single basic block may branch out to many others. The process of ensuring that these transitions between basic blocks are legal is called Control Flow Checking. A legal transition is defined as one that is allowed by the control flow graph determined at compile time before the program is run.

At compile time, a graph is generated showing all legal branches. Each basic block is represented by a node. A unique signature ($s_n$) is assigned to each basic block. Along with this, a signature difference ($d_n$) is assigned to each basic block, which is calculated by taking the bit-wise XOR ($\oplus$) of the current block and its successor. When the program is run, a run-time signature tracker ($G_n$) is updated with the signature of the current basic block. When the program branches to a new basic block, the signature tracker is XOR'd with the signature difference of the new block:

$$G_n \oplus s_n = d_n$$

Because the XOR operation can undo itself, the result should equal the signature of the current block. If it does not, then a control flow error has been detected.

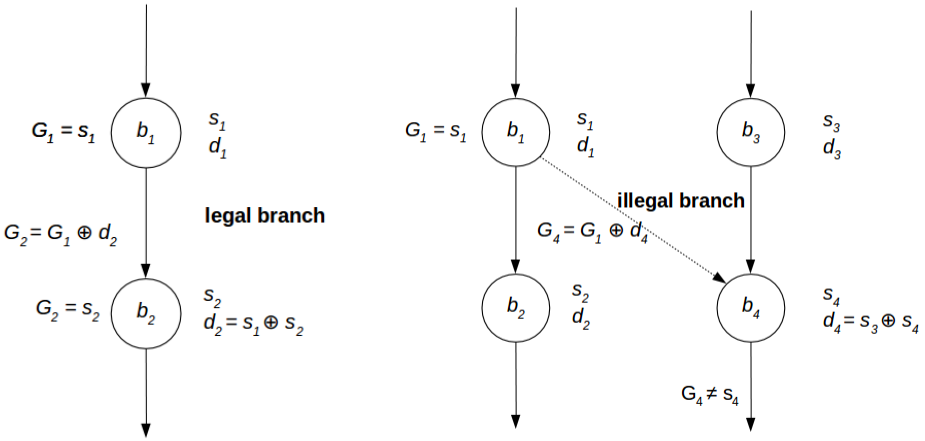


Figure 1: correct vs. incorrect branching

### 3.1 Branch Fan In

There is a danger when dealing with dense control flow graphs that there will be a configuration as seen in Figure 2:

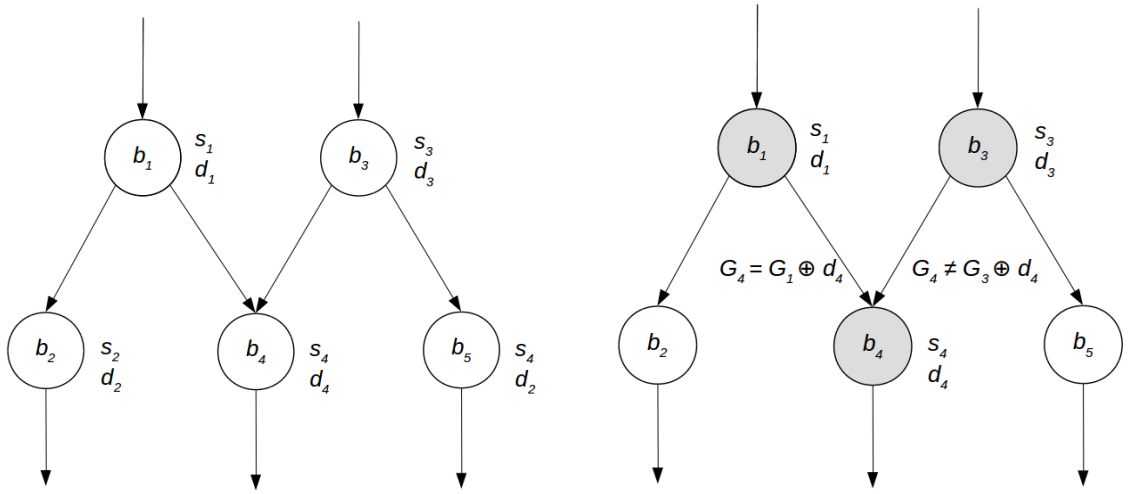


Figure 2: branch fan in problem

If $b_1$ and $b_3$ are assigned the same signature, then there will be no issue branching to $b_4$. However, this opens up the possibility for illegal branching from $b_1$ to $b_5$ without being caught. If all signatures are generated randomly, without any duplicates, then $b_4$ will register correct branching from either $b_1$ or $b_3$, but not both.

This necessitates the addition of the run-time signature adjuster. $(D_n)$ This is an additional number that is calculated at compile time for each basic block, then updated as the program executes. It is used to adjust for the differences created by this branch fan-in problem.

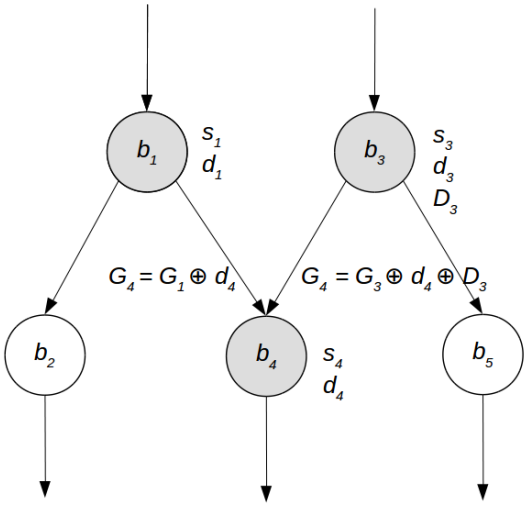


Figure 3: run-time signature adjuster

In the case of the branch from $b_1$ to $b_4$, the signature adjuster will be 0. In the case of the branch from $b_3$ to $b_4$, the signature adjuster will be

$$D_3 = s_3 \oplus d_4 \oplus s_4$$

such that

$$G_4 = G_3 \oplus d_4 \oplus D_3$$

# 4 Modifications

Although the algorithm described above is very robust, there were some instances where it does not perform correctly. If a node has two successors which are themselves both branch fan-in nodes (as in figure 4), the algorithm will correctly assign a signature adjuster value for one branch, but not for the other.

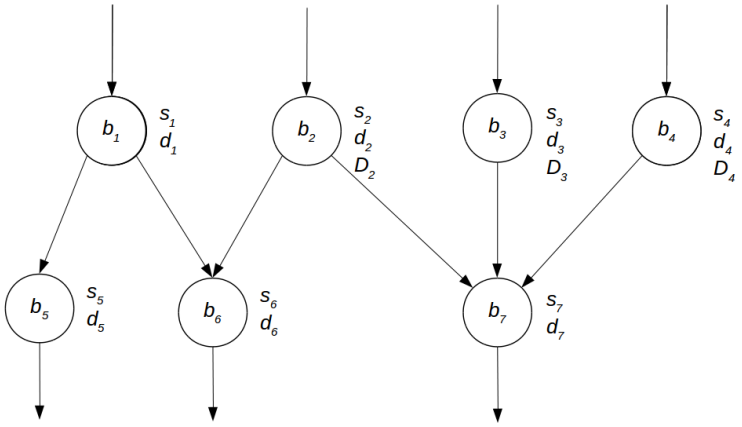


Figure 4: multiple successors with branch fan-in

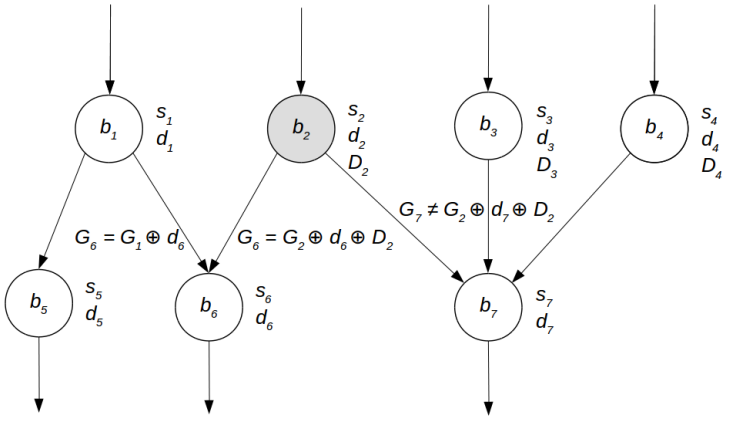


Figure 5: run-time signature adjuster error

To solve this problem, we determined to insert an extra basic block to act as a buffer. This would go between the predecessor with the invalid signature adjuster and the successor that is the branch fan-in node (see Figure 6) It would contain no instructions other than those that verify proper control flow. Because this buffer block would only have one predecessor, it would not need to use the signature adjuster, whatever the value might be. The value for $D_8$ for the buffer block would be determined to allow correct branching to the successor node.

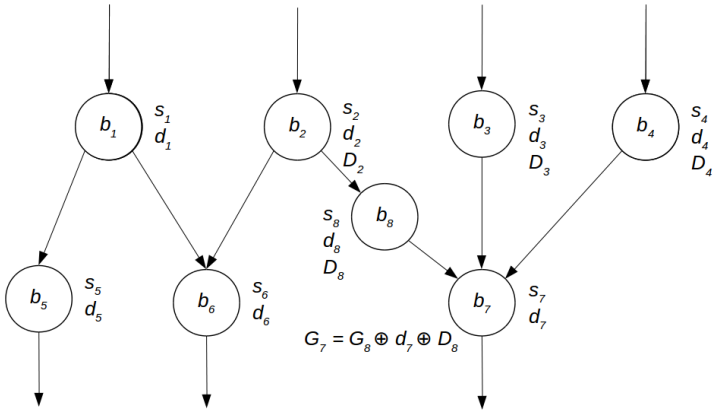


Figure 6: run-time signature adjuster error

# 5 Implementation

We implemented this algorithm using LLVM. It was implemented as a pass that the optimizer runs before the back-end compiles the assembly into machine code. This particular implementation worked very well with the algorithm, because LLVM automatically splits its programs into basic blocks. One of the challenges this presented was compiling for a 16-bit microprocessor. In order to save space, the signatures were generated as unsigned 16-bit numbers. This gives 65,535 possible signatures to use, which far surpasses the number of basic blocks you could fit in such a small memory space as we had on our device.

To deal with the multiple fan-in successor problem mentioned above, we ran the signature generation step as normal. Then we checked the entire graph to see if there were any mismatched signatures. If there were, we inserted a buffer block to deal with that problem and updated the surrounding blocks to match the new block.

To implement the control flow checking, we inserted a set of instructions at the beginning of each basic block to do the XOR operation specified above. We also inserted instructions at the end of each block to update the run-time signature tracker to be the signature of the block about to be left.

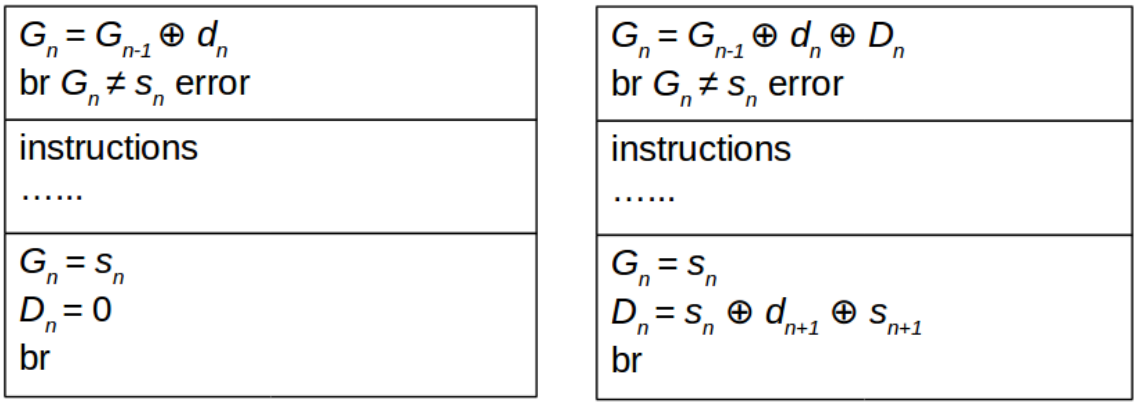


Figure 7: inserting instructions into basic blocks

One of the optimizations we used was to only insert the extra XOR operation when $D_{n-1}$ was $\neq 0$. This is one reason why the buffer block fix worked.

# 6 Results

[insert results here]

# 7 Conclusion

[insert awesome text here]