FFXE: Dynamic Control Flow Graph Recovery for Embedded Firmware Binaries

Ryan Tsang, Asmita, and Doreen Joseph, *University of California, Davis;* Soheil Salehi, *University of Arizona;* Prasant Mohapatra and Houman Homayoun, *University of California, Davis*

https://www.usenix.org/conference/usenixsecurity24/presentation/tsang

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

FFXE:
Dynamic Control Flow Graph Recovery for Embedded Firmware Binaries

Ryan Tsang
University of California, Davis

Asmita
University of California, Davis

Doreen Joseph
University of California, Davis

Soheil Salehi
University of Arizona

Prasant Mohapatra
University of California, Davis

Houman Homayoun
University of California, Davis

Abstract

Control Flow Graphs (CFG) play a significant role as an intermediary analysis in many advanced static and dynamic software analysis techniques. As firmware security and validation for embedded systems becomes a greater concern, accurate CFGs for embedded firmware binaries are crucial for adapting many valuable software analysis techniques to firmware, which can enable more thorough functionality and security analysis. In this work, we present a portable new dynamic CFG recovery technique based on dynamic forced execution that allows us to resolve indirect branches to registered callback functions, which are dependent on asynchronous changes to volatile memory. Our implementation, the Forced Firmware Execution Engine (FFXE), written in Python using the Unicorn emulation framework, is able to identify 100% of known callback functions in our test set of 36 firmware images, something none of the other techniques we tested against were able to do reliably. Using our results and observations, we compare our engine to 4 other CFG recovery techniques and provide both our thoughts on how this work might enhance other tools, and how it might be further developed. With our contributions, we hope to help enable the application of traditionally software-focused security analysis techniques to the hardware interactions that are integral to embedded system firmware.

1 Introduction

Embedded systems are heavily relied upon in nearly every aspect of our daily lives, from the keyboards we use on a day-to-day basis, to the programmable logic controllers that coordinate complex manufacturing processes, to the medical imaging devices found in our hospitals. For many of these applications, system safety and security are of critical importance. A crucial component of realizing this is firmware verification and vulnerability analysis, which is unfortunately lacking in many modern systems [45]. As a specialized form of software, firmware images can theoretically be analyzed with traditional software analysis techniques. However, this is often difficult in practice due to fundamental differences between firmware and software. Nonetheless, if software analysis techniques can be reliably adapted to firmware, there could be significant improvements in the security, trust, and assurance of embedded systems.

In this paper, we focus on Control Flow Graph (CFG) recovery as an important enabling step for applying more complex software analysis to monolithic bare-metal binary firmware images. We present the Forced Firmware Execution Engine (FFXE), which leverages the original dynamic forced execution algorithm [48] to enable the resolution of indirect branches whose targets are determined asynchronously. Such branches often arise from callback function registration, a programming archetype that occurs ubiquitously in embedded systems development. To accomplish this, we introduce several additional stages to the original technique that allow us to resume execution from instructions that access volatile memory when a potential branch target is written to the same location. Our prototype is implemented as a manager class designed to handle the additional context information needed to track volatile memory accesses, as well as execution thread state. Our code artifacts are publicly available on GitHub. [1]

Our paper is outlined as follows: the rest of this section gives a high-level overview of the motivations for this work and a summary of our contributions; Sections 2 and 3 provide further background on CFG recovery techniques and related work; Section 4 details the design and implementation of our technique; in Sections 5 and 6 we present and discuss our results; and finally in Sections 7 and 8 we discuss future work and make our final remarks.

A CFG is a graph representation of all of the possible execution paths that might be taken through a program [3]. The vertices of a CFG are called basic blocks, which are contiguous sets of sequentially executing instructions that have a single entry point and exit point. Exit points are control flow instructions that can alter the value of the program

[1] https://github.com/rchtsang/ffxe

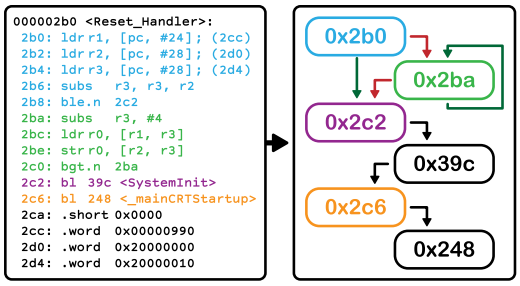

Figure 1: CFG Recovery Example.

counter, typically function calls, function returns, or conditional branches. The edges of the CFG represent the branch targets of the basic block's last instruction and are therefore directed. A small example of a CFG for a single function can be seen in Figure 1. CFGs are valuable structures for program analysis and are used in many applications like verification [5, 8, 15, 18, 23, 47], data flow analysis [2, 43], and malware detection [1, 10, 19, 27, 28, 30, 32]. As embedded and cyber-physical systems continue to grow in ubiquity, such techniques have become increasingly necessary for ensuring the security of the firmware that drives them.

As already mentioned, it is often difficult in practice to apply software analysis techniques to the firmware. This is because, by nature, firmware code is strongly coupled to the hardware that it is written for. While software usually abstracts hardware interactions through the operating system and concerns itself only with application behavior, bare-metal firmware must manage both application and hardware, since hardware dependencies are often directly referenced through memory-mapped input/output (MMIO). Moreover, MMIO references are usually linked to hardware events that have a direct effect on firmware control flow, making execution paths unpredictable based on syntax alone.

In particular, embedded systems tend to interact with the physical environment through hardware peripherals like sensors and actuators. Thus, firmware tends to have a high degree of asynchronicity, as interrupts and interrupt service routines (ISRs) often play critical roles in responding to external events triggered by peripherals. An ISR is typically a programmer-defined function that gets registered to a particular event. When that event occurs, an interrupt is triggered that stops the current thread of execution, switches context to the registered ISR, and executes it. When the ISR thread finishes executing, the processor context is restored to the point of interrupt and continues with the main thread of execution. This complicates many analyses because values can change unpredictably from a given thread's point of view. These values are considered *volatile* and may need to be handled differently depending on their underlying hardware dependency.

These differences between software and firmware limit the effectiveness of existing CFG recovery techniques, as existing techniques are designed primarily for serial software and fail to account for asynchronicity that arises due to hardware dependency in firmware. In this paper, we address some of these challenges by building upon existing techniques to enable greater support for firmware CFG recovery. Our contributions can be summarized as follows:

- We present a portable dynamic algorithm for CFG recovery, designed specifically to handle indirect branches that are affected by interrupt-based asynchronicity (volatile memory).
- We implement and evaluate a prototype of our algorithm, the Forced Firmware Execution Engine (FFXE).
- We demonstrate FFXE's effectiveness at resolving indirect branches whose targets are dependent on asynchronous memory writes (registered callback functions).
- We compare our prototype's performance against several CFG recovery techniques, including a reimplementation of the original dynamic forced execution algorithm.

2 Background

2.1 Control Flow Graph Recovery

Naturally, a complete control flow graph (CFG) can be constructed trivially from the syntax of the program's source code, as all paths are inherently enumerated by the programmer. However, while such CFGs are useful in many cases, due to compiler optimizations in compiled languages, they do not always accurately reflect the actual control flow of the compiled binary. As such, there has been significant research effort given to recovering control flow graphs directly from binaries [16]. This amounts to a graph exploration problem, in which the graph information is embedded in the firmware binary, and graph vertices and edges are only revealed as an exploration algorithm is executed. A toy example of CFG recovery is shown in Figure 1. The figure demonstrates the identification of basic blocks in ARM Thumb disassembly, as well as the edges between those blocks that indicate control flow transfers due to conditional branch or function calls. While straightforward in principle, there are many challenges that make it difficult to recover a complete and accurate CFG. Existing research on CFG recovery deals with many specific issues, but perhaps the most fundamental one is the problem of *indirect jumps*. These are control flow instructions whose branch targets are not fixed–i.e. dependent on the value stored in a register or memory location. Since this information is only known at runtime, it is difficult to guarantee perfect CFG recovery for software that contains indirection.

Since their inception, the most common CFG recovery methods have been largely static, relying purely on the analysis of program disassembly without runtime information like processor or memory state [39]. The typical approach to static CFG recovery is to scan the entire program binary for jump

and branch instructions whose targets can be resolved to begin recovering basic blocks. When a target is found, the recovery engine simply continues to disassemble contiguous instructions until the branch instruction that signifies the end of a block is reached. This is continued until no new blocks can be found [40, 42]. However, fundamentally, static recovery techniques suffer from the problem of indirect jumps—branches that are dependent on memory and register state. Significant efforts have been made to address this problem and many improvements have been applied to static techniques which can help offset this problem [50], but the fundamental limitation remains. By introducing dynamic execution, we can theoretically resolve the indirect jump issue, as runtime information can be extracted to resolve indirect jumps. As such, there have been significant research efforts to build hybrid and fully dynamic recovery engines which can yield more complete, correct, and accurate CFGs [11]. These efforts have ranged in their approach from pure emulation, to symbolic execution. Moreover, the extracted runtime information can be used to enhance other techniques and enable more complete analyses, such as execution frequency annotation, on the resulting CFG [1].

2.2 Dynamic Forced Execution

Introduced by Xu et al. in 2009, forced execution is a dynamic technique for recovering CFGs predicated on the observation that the calculation of indirect jump targets is usually independent of intermediate program states, and postulates that an accurate CFG can be recovered by exploring both directions of every encountered conditional branch [48]. The original optimized algorithm for the Forced Execution Engine (FXE) is presented in Algorithm 1 In FXE, **(1)** the program is initialized in a virtual environment and an empty CFG is instantiated. **(2)** The engine begins executing the program at its entry point and an exit point is inferred from the entry point's value and the code section size. **(3)** Path exploration is directed with a coloring system for conditional branches. When a conditional branch is encountered, its location is saved and uncolored, indicating that neither path has been explored yet. **(4)** The engine then saves the current processor and memory state, as well as the target address of the branch path not taken. **(5)** The engine then colors that branch gray and continues down the execution path that is consistent with the emulated branch condition. **(6)** When the current path terminates, either because the engine has encountered a known basic block, or because the program itself has terminated, the engine restores context to the nearest gray branch, colors it black, and continues down the unexplored path. This results in a depth-first exploration pattern that terminates when all branches have been marked black. However, the original authors implement further optimizations to ensure the timely termination of their algorithm.

In order to avoid long-running or infinite loops, the engine assigns each basic block an execution quota that determines whether it should be skipped or not. On the first encounter, a block is given a quota of 1 and the quota is decremented after execution. If a block is encountered that ends with an indi-

Algorithm 1: Dynamic Forced Execution

```
Output: CFG cfg
Input: Executable exe
cfg = NULL; current = NULL; block = NULL;
  ip = get_instruction_pointer();
  branches = [];

while true do
    while block = get_block(ip) do
        if !(exe.EntryPoint ≤ block.pc < exe.ExitPoint) or (block.quota
          <= 0)) then
            try
                ip = execute_block(block);
            catch (exception)
                error_handler(ip, block, exception);
            continue;
        connect_block(cfg, current, block);
          current = block;
        while inst = get_instruction(block, ip) do
            if (inst.type == ConditionalBranch) and
              (find_branch(inst) == NULL) then
                branch = get_branch(inst);
                if branch_is_taken(branch) then
                    branch.next_ip = ip + inst.length;
                else
                    branch.next_ip = get_target(branch);
                branch.state = gray;
                  save_context(branch);
                  add(branch, branches);

            else if (inst.type == CallBranch) and (find_branch(inst)
              ≠ NULL) then
                branch = get_call_branch(inst);
                    get_return_points(branch, cfg);  // forward
                    reachability
                branch.next_ip = ip + inst.length;
                  branch.state = gray;
                  save_context(branch);
                  add(branch, branches);

            if (inst.type == IndirectBranch) and (find_target(inst) not
              in known_targets(inst)) then
                add_target(inst);
                    increment_quotas(current, cfg);  // backward
                    reachability
            try
                ip = execute_inst(inst);
            catch (exception)
                error_handler(ip, block, exception);
    if branch = get_last_gray_branch() then
        load_context(branch);
          ip = branch.next_ip;
          branch.state = black;
          current = get_branch_block(branch);

        if branch.type == CallBranch then
            add_return_points(branch,current.predecessors);
    else
        break

resolve_blocks(cfg);
  return cfg;
```

rect branch, and the branch target has not been seen before, then after executing it the engine will increment its quota and the quota of every block leading up to it within the current function. A block is thus only executed if it has a non-zero quota, while still allowing for the future discovery of other indirect branch targets. However, an additional consideration needs to be addressed to preserve CFG edge completeness. To circumvent the loss of function fall-through edges–which may occur when a function that has already been explored is called from a new location, resulting in termination before returning–function calls are treated similarly to conditional branches. The call branch is colored gray and added to the branch list with the saved fall-through path context (i.e. the return address). To capture the return edge from the function to the fall-through path, forward reachability analysis is performed on the function to determine the possible return points, which are all added as return edges for the fall-through path. An exception handler is also implemented to allow for graceful recovery from crashes that may arise from exploring the contradictory path at conditional branches.

While the forced execution method appears to be quite effective at recovering CFGs based on coverage results of the reported benchmark, the algorithm does have its drawbacks. Particularly, in its presented form, it has only limited support for recovering unconnected portions of the CFG that do not exist as part of the main thread of execution but may be executed asynchronously, such as those representing interrupt service routines (ISR). The authors rely on manually enumerated instruction patterns in order to detect interrupt handler registration, after which the forced execution engine can be invoked on the registered interrupt [48], where event and interrupt-driven programming is prevalent, the manual enumeration of patterns for handler registration patterns is impractical since there are many ways that an ISR might be registered. Moreover, forced execution neglects the role that volatile memory addresses may play in resolving the indirect jumps, and does not handle cases where indirect jumps targets are contingent on volatile data. In order to adapt forced execution to event-driven firmware, we relax some of the assumptions that FXE makes about the independence of program states, and augment to FXE algorithm to account for volatile memory and interrupt handlers.

3 Related Works

Indirect jump resolution in CFG recovery remains an ongoing topic of research. While many efforts have been made to resolve indirect jumps in a purely static fashion [6, 7, 14, 17, 21, 22, 29, 37, 37, 44], many of the more recent approaches to this problem leverage dynamic analysis techniques [20, 32, 36, 38, 39, 42, 49, 50]. In this section, we outline some of these works, focusing on methods that leverage dynamic analysis techniques.

Nguyen et al. [32] propose a hybrid approach that combines static and dynamic analysis for CFG generation from binaries. Symbolic execution is used in the static analysis phase until an indirect jump or function call is encountered. Test cases are then generated to cover all execution paths, and dynamic analysis is used to execute the test cases to resolve jump targets. Static analysis is then invoked to analyze new targets, and the process is repeated in this alternating manner until no new targets are found.

In a continuation of work on forced execution, Peng et al. [36] introduce *X-Force*, a binary analysis engine that applies forced execution for a small set of conditions to explore paths and expose a binary's behavior. Dynamic binary instrumentation is used to monitor the concrete program state of the target binary. *X-Force* is used for three different applications of CFG recovery, malware analysis, and reverse engineering, thereby demonstrating the potential of force executing binaries for security purposes. However, You et al. note that because *X-Force* is a heavy-weight engine and therefore suffers from scalability issues, it is impractical to use [49]. They improve upon it with PMP, a practical forced-execution engine that instead avoids tracking individual instruction and on-demand allocation using a buffer pre-allocation scheme that prevents pointer dereferencing exceptions and probabilistically avoids state corruption.

A popular framework for dynamic binary analysis, `angr` [42] offers two separate recovery algorithms: *CFGFast* and *CFGEmulated*. *CFGFast* uses static analysis for CFG generation, sacrificing the graph recovery accuracy for speed and coverage, employing standard static recovery techniques such as function prologue matching and recursive disassembly. *CFGEmulated* uses a combination of dynamic forced execution, symbolic execution, and backward slicing, prioritizing completeness in resolving indirect jumps, resulting in a more accurate CFG at the cost of execution time. The first stage of the algorithm uses forced execution to seed further CFG recovery by adding basic blocks and their *direct* jumps only to the CFG under construction. To ensure that *indirect* jumps are resolved correctly, `angr` does not make the assumption that indirect branches are independent of intermediate program states. Thus, all indirect jumps discovered with forced execution are stored for later analysis. `angr` then uses *symbolic execution* on the path to an indirect jump, utilizing a constraint solver to retrieve possible jump targets. Any unresolved jumps are further analyzed in a context-sensitive manner by creating a backward slice of the program starting from the unresolved branch that includes all call contexts. Because `angr` is designed specifically for binary analysis, many of its techniques can be applied to ARM-based firmware out-of-the-box. However, it is fundamentally designed for general-purpose software, making it unclear how effective it is for firmware, which exhibits many non-standard programming patterns.

Recently, Zhu et al. [20] have proposed another hybrid recovery technique for x86 binaries that uses static analysis recover an initial CFG, then employs coverage-based gray-box

fuzzing to generate test cases that are used to resolve indirect jumps. The technique is further developed in a later work using directed gray-box fuzzing [50] on C/C++ programs at the source code level. Hence, while interesting, neither techniques are suitable for binary analysis of firmware images, which are almost never in x86, nor have access to source code.

4 Design and Implementation

At a high level, we propose a forced execution-based algorithm that takes memory accesses into account in order to recover registered callback functions. In our algorithm, we first apply forced execution to each known ISR entry point in order to determine what addresses may be accessed asynchronously. Once this scan is complete, we begin forced execution from the main entry point, watching for memory writes to addresses that overlap with those accessed by ISRs. When such accesses are detected, we restore the corresponding ISR execution state, overwriting the accessed memory location with the value of the write. When the values written are registered function addresses, this allows the emulator to correctly resolve subsequent indirect branches that rely on that value. However, before we discuss our algorithm in depth, it makes sense to first discuss the reimplementation of the original Forced Execution Engine (FXE), as some minor changes to the original algorithm must be made for use with our chosen emulator. As our algorithm relies on portions of the original, these modifications will be carried forward into our algorithm's implementation.

4.1 Reimplementing FXE

As the method of forced execution requires dynamic information, we require an emulator for the instruction set of the target embedded system. For this, the Unicorn emulation framework [13] was an ideal choice, as it provides a more flexible interface to the powerful JIT binary translator used in the Qemu emulator, and is a good platform for adapting forced execution to other architectures. We choose Python to implement the proof of concept, as it is a widely adopted language and easy to work with. Though Python is generally slower and allocates more memory at runtime than other languages, our priority is proof-of-concept, so this is acceptable. We also rely on the Capstone disassembly framework [33] to identify instructions. However, because of some unfortunate bugs in the Capstone and Unicorn frameworks, some of the memory decoding and access checking functionality became unreliable. Thus, decoding crucial memory and branch instructions required custom structures to parse opcodes and parameters.

Since the original forced execution algorithm was implemented using neither Python nor Unicorn, we first reimplement the original forced execution engine (FXE) in Python and Unicorn for later comparisons. Our version of the original includes the block execution quota system, as well as the associated backward and forward reachability steps necessary to deal with the complications that inevitably arise when blocks are not allowed to execute. We do not include a separate construct for error handling, as exceptions can be handled on a case-by-case basis using Python's built-in `try-except` syntax.

Unicorn already encapsulates an emulator and exposes emulator state information primarily through a hook/callback API, so the algorithm could not be implemented in the same form as the original Algorithm 1. We instead implemented FXE piecewise, in separate callback functions for blocks and instructions. These callbacks are executed when the emulator encounters a new Qemu translation block and the next instruction, respectively. Hence, while the forced execution algorithm remains fundamentally the same, different portions of the algorithm must be written into the callback function suitable to the operation being performed. As mentioned, this was further complicated by bugs in the Unicorn and Capstone code base which were leading to the unreliability of hooking memory accesses and decoding of branch and memory instructions. We therefore defined custom `ctypes` structures for each branch, load, and store instruction type to guarantee proper behavior for the algorithm. This was done for the most common types of single load and store instruction, but not all, so there is some chance of incompleteness.

4.2 Implementation

We augment the original FXE algorithm by introducing some platform-aware modifications, specifically targeting ARM-based microprocessors for our prototype, since ARM-based systems hold a large share of embedded systems market. Modern ARM architectures come in 3 variants: Cortex-A for general-purpose computing, Cortex-M for efficient computing in constrained resource environments, and Cortex-R for performance computing with real-time requirements. Of the 3, Cortex-M is most frequently used for embedded systems and microcontrollers, though Cortex-R is not uncommon, and is the successor to older architectures that employed both ARM and Thumb modes of the instruction set architecture (ISA), such as ARM7TDMI. We design our prototype primarily for Cortex-M, though we have also implemented some support for ARM7TDMI and Cortex-A/R, which we discuss in later sections.

4.2.1 Locating ISR Entry Points

Before we can run FFXE on a given firmware, we must first determine its program entry points: the main entry point, and the ISR entry points. The Cortex-M architecture family implements hardware interrupts using the Nested Vector Interrupt Controller [4]. While we will not delve into the details, it suffices to know that hardware events are linked to an entry in the NVIC's vector table, which contains the address of a call-

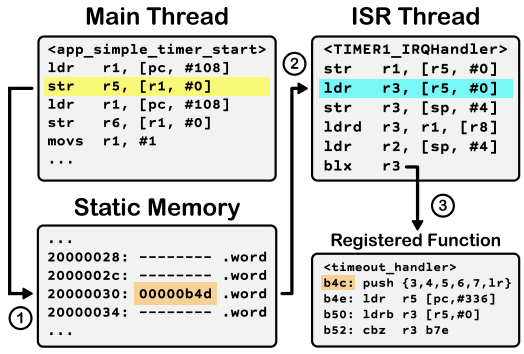

Figure 2: Registered Function Example.
(1) The main thread writes the registered function address `0xb4d` to a static address known to the API callback. (2) In the ISR thread, the API callback reads the registered function address into a register. (3) The API callback then calls the registered function by jumping to the address in the register.

back function. The vector table is a sequence of consecutive words that correspond to the location of a callback function or interrupt service routine (ISR) that will be invoked if its corresponding hardware event occurs. Particularly important is the Reset Handler entry, which is the first entry of the vector table. As defined by the Cortex-M architecture, on cold reset or startup, the processor will always initialize the program counter from the fixed address `0x4`, which is also the default location of the Reset entry of the vector table. While the vector table can be relocated at runtime, `0x4` will still contain a valid pointer, which we can take to be our main entry point if the firmware image base address is still `0x0`. If this is not the case, prior work in FirmXRay [46] has been done on statically locating relocated vector tables, which we can utilize to attempt to locate a valid base address.

In general, the first set of vector table locations are used for exception handling and other privileged use-cases. However, starting from offset `0x40` in the vector table, generic interrupt handlers can be registered [24] per vendor hardware specification. For example, the nRF52 series of microcontrollers maps specific sets of peripherals to each generic ISR entry point [41]. The vector table provides us with entry points for starting forced execution on these handlers, but the quality of the resolved blocks will still be poor if the state of the emulator's volatile memory locations does not accurately reflect the expected state that the handler should operate in. This is due to the fact that these handlers are typically responsible for invoking user-registered callbacks, which means the functions must be registered in the emulator's memory at the time of execution if they are to be recovered. This also means that the state of volatile memory locations must be taken into account to resolve indirect jumps that rely on volatile data.

4.3 Designing for Volatile Memory

As previously stated, volatile memory locations play a prominent role in embedded systems programming. While "volatile" can also refer to the permanence of memory with respect to power cycling. Herein, we use the term to refer to memory locations whose data may be altered asynchronously—the same way the`volatile` C keyword is used. In embedded systems, a volatile memory location typically corresponds to a memory location that is accessed by both the main thread of execution and an interrupt service routine, or to a memory-mapped hardware peripheral register. This is especially relevant for interrupt service routines, as it is a common programming pattern to *register* functions to particular hardware events using an application programming interface (API). Such APIs, like those in the nRF52 software development kit [34], are typically implemented in such a way that some default handler is defined for each possible hardware event, which then invokes a *callback function* that was registered at API initialization. An example of this is shown in Figure 2. This means that there is indirection associated with interrupt events, which poses a significant challenge for CFG recovery, as most static methods will have no way to resolve an indirect jump whose target is determined dynamically in a separate execution thread. This is the primary issue we attempt to tackle in this work.

4.3.1 Modifying Forced Execution

To address the aforementioned challenges, we propose the following algorithm for volatile-aware forced execution:

1. Initialize the emulator's memory state by pre-executing the reset handler, which is responsible for copying pre-initialized global variables into RAM. This is done before forced execution begins because it typically involves a loop that may not be fully executed during forced execution due to the block execution quota system.

2. Do a forced execution pass on each of the ISR function addresses in the vector table, excluding the main entry point. During each of these passes, the engine records each memory access as a volatile memory access.

3. If the volatile memory access was a read, the engine creates a backup of the emulator's CPU and memory context associated with the address and saves the vector table entry that it was descended from. This will allow it to resume execution from that point when the volatile address is written in the main thread of execution.

4. After a pass is conducted on each vector table entry, the quotas on all ISR blocks are reset and the engine begins forced execution at the main entry point. As it executes portions of the original algorithm along the main thread, it also watches for accesses to volatile addresses and logs any memory writes, volatile or otherwise, to a dictionary.

5. Whenever a volatile memory address is written, the engine resumes execution from any reads to the same address that occur in a different thread context, modifying the memory state to reflect that at the time of writing

so that the written value has a chance to influence any indirect jumps that may rely on it.

6. If any new blocks are discovered in an ISR thread, the engine updates its record of volatile memory locations with any additional memory accesses that are encountered, saving and restoring execution contexts as necessary if it is found that a previously written address is now identified as volatile.

7. Additionally and optionally, the engine queues additional forced execution passes on ISR entry points under 2 conditions: **(a)** when it is recognized that interrupts have been enabled for a given peripheral via a memory-mapped register write, and **(b)** when wait-for-interrupt (WFI) or wait-for-event (WFE) instructions are encountered in the main thread. This is done to increase the odds of capturing as many possible memory states that might be valid under preemption, so as to improve the odds of finding valid indirect execution paths.

We acknowledge that our technique exhibits significant coupling with hardware and ISA, particularly with respect to the second and last step in our algorithm. In step 2 we rely on the Cortex-M definition of the vector table to locate ISR entry points; however, the concept of the vector table is not specific to the Cortex-M architecture, and nearly all common architectures for embedded systems implement a similar scheme to facilitate interrupt handling. Moreover, in step 7, we note that the precise memory address that enables interrupts is completely platform dependent, and the WFI and WFE instructions are specific to the ARMv7e-M ISA. However, note that the underlying concept of this step can be generalized to other hardware platforms and architectures. Note further that this step is not integral to the algorithm, as it is included primarily to further cover the search space for indirect execution paths–paths that are contingent on indirect jump targets. The core contribution is the idea of resuming execution from volatile memory accesses in order to resolve indirect jump targets that are contingent on asynchronous writes to those locations.

4.3.2 Edge Cases

In this section, we discuss the various edge cases that have been considered during implementation. As one might expect, there are a significant number of edge cases that arise as a result of forcibly executing indirect jumps from invalid execution states. Moreover, because compiler optimizations can significantly change the layout of the original source code, there are additional edge cases to consider where typical conventions may have been violated. Additionally, the ARMv7e-M instruction set implements some instructions that have unique effects and must be handled specially.

Encountering Invalid Addresses: During the course of forced execution, it is very common that an invalid basic

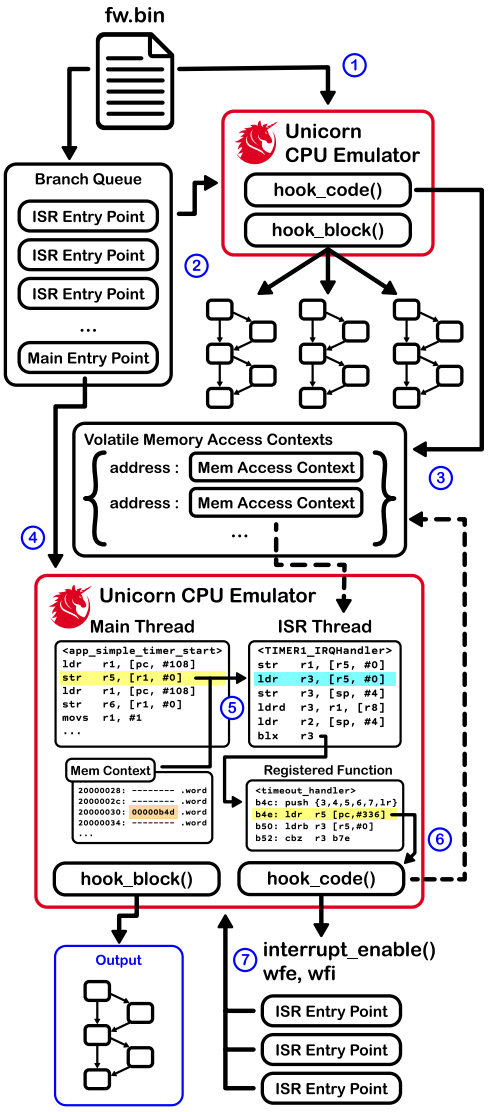

Figure 3: FFXE Algorithm Diagram

block will be encountered as a result of either an indirect jump from an inconsistent processor state or a fall-through edge that does not correspond to an actual function return. We observed that these invalid block locations tend to fall into one of three categories: (1) the 0 address and vector table entries; (2) data that has been interspersed within code sections; and (3) non-instruction regions like separate data sections, empty code regions past the edge of the firmware image, and hardware-mapped memory regions. The first and last categories are typically a result of invalid indirect jumps and are dealt with by halting execution if an address within those ranges is encountered. To deal with invalid attempts to fetch interleaved data, we refer to the memory log to determine if the address has been read from before. If it has, we assume that the address corresponds to data and halt the execution, allowing the engine to continue on to the next path to explore. This category of invalid locations tends to be encountered as

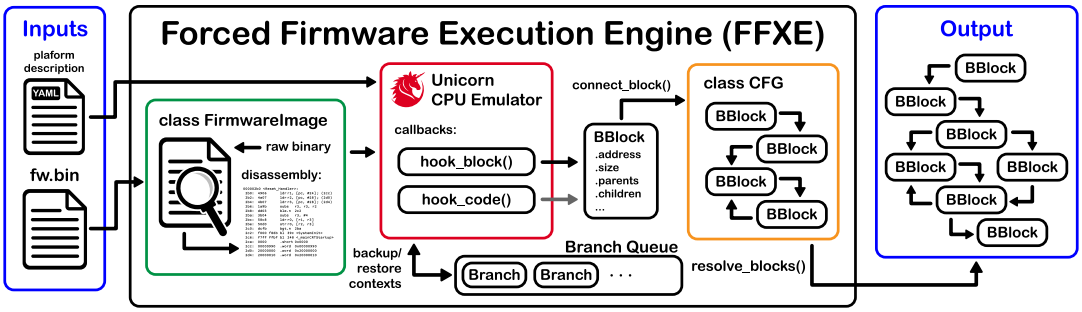

Figure 4: FFXE Software Architecture Diagram.

a result of an invalid fall-through edge, i.e. one that does not correspond to a returning function call, as in the case of tail call optimized functions. In addition, we also check that every instruction in a new translation block can be successfully disassembled, terminating if any invalid instruction is encountered. In each case where we terminate execution, we mark the invalid block for deletion, which occurs alongside the resolution of overlapping blocks at the end of the algorithm. In doing this, we prioritize the validity of our reconstructed CFG, attempting to prune those blocks and edges which we believe do not appear in the true CFG. However, in rare cases, this can remove a block that is actually a known target of a direct jump. In this case, we want to prevent the deletion of the block and edge. Therefore, we also keep track of explicitly defined jump targets and protect them from removal, forcing the engine to explore the defined targets regardless of the address location and preventing over-pruning.

Detecting Inlined Functions: It appears that tail call functions result in more than just invalid fall-through edges. They also complicate the detection of inlined functions, since an inlined function invoked via tail call may violate expectations if it is done without explicitly using a branch-and-link instruction. For instance, this may be done by overwriting the return address of a function address before a `pop` instruction is invoked, calling the next function implicitly similar to return-oriented programming, or making an unconditional branch straight to the inlined function without linking. While it is impossible to guarantee the detection of inlined functions under all circumstances, we are able to guess their presence if they are called more than once from separate locations. Since each block is labeled with the address of the function they are believed to belong to, we can detect if a block has been executed before from a different function context. This is because, in our implementation, blocks inherit their function label unless a branch-and-link instruction gives it a new one or a function return is detected based on the call stack return address. When an existing block's function label mismatches the current context, we assume that it is the start of its own function and relabel it, and any of its successors with the same address. This solution is not formally proven. In particular, the propagation of function labels may be troublesome, but in practice, it does not seem to cause undue pruning.

Thumb IT Instruction: This instruction is used to define conditional blocks for otherwise unconditioned instructions, allowing the programmer to conditionally execute several sequential, individual instructions based on a single condition code [25], this can complicate the forced execution algorithm if taken to be separate control flow paths since it would break the conditional block into several separate basic blocks. It turns out that `Unicorn` treats these blocks as a single translation block, so we are incapable of resolving this aspect of the control flow. Hence, we resign ourselves to dealing with conditional blocks only in the event that they end in a branch instruction, making that a conditional branch that requires a context split.

Table Branch Instructions: The ARMv7e-M ISA also implements the table branch instructions `tbb` and `tbh`, which are essentially the ISA's version of a jump table [25]. These instructions enable PC-relative forward branching based on a table of single-byte or single halfword offsets, respectively. A base register is used to indicate the start of the table, which can be the PC, allowing the table to be interleaved with instructions in the code section. The branch target is twice the value of the table entry. Under forced execution, the natural way to deal with this is to follow every unique branch path in the table, looping through the entries and adding each one to the exploration queue until we reach the end of the table. However, as far as we can tell, there is no hard limit on the size of the table, so we rely on empirical observations and assumptions made in line with Cojocar et al. [12] to decide when to stop adding entries, potentially resulting in long-running loops or lost coverage. We first observe that table branch instructions are almost exclusively compiled from `switch` statements, which results in a short data section inserted between the table branch instruction and the compiled `case` statements. This means that as we loop through the entries if we encounter a memory address that corresponds to a previously calculated branch target, we have hit the end of the table and can terminate the loop. This strategy proves effective in the cases we have observed, but to avoid overestimating the number of jumps, we also introduce termination conditions for several other scenarios: if the target is outside of a valid

Table 1: Registered Function Resolution Comparison of CFG Recovery Methods.

Firmware		angr_emu	angr_fast	**ffxe**	fxe	ghidra
gpiote	-O0	0/0/1	0/1/1	**1/1/1**	0/0/1	0/1/1
	-O1	0/0/1	0/1/1	**1/1/1**	0/0/1	0/0/1
	-O2	0/0/1	0/1/1	**1/1/1**	0/0/1	0/0/1
	-O3	0/0/1	0/1/1	**1/1/1**	0/0/1	0/0/1
i2s	-O0	0/0/4	0/4/4	**9/4/4**	0/0/4	0/4/4
	-O1	0/0/4	0/4/4	**10/4/4**	0/0/4	0/3/4
	-O2	0/0/4	0/4/4	**12/4/4**	0/0/4	0/3/4
	-O3	0/0/4	0/4/4	**12/4/4**	0/0/4	0/3/4
saadc	-O0	0/0/2	0/2/2	**4/2/2**	0/0/2	0/2/2
	-O1	0/0/2	0/2/2	**4/2/2**	0/0/2	0/2/2
	-O2	0/0/2	1/2/2	**4/2/2**	0/0/2	0/2/2
	-O3	0/0/2	1/2/2	**4/2/2**	0/0/2	0/2/2
simple_timer	-O0	0/0/2	0/2/2	**2/2/2**	0/0/2	0/2/2
	-O1	0/0/2	0/2/2	**2/2/2**	0/0/2	0/1/2
	-O2	0/0/2	1/2/2	**2/2/2**	0/0/2	0/1/2
	-O3	0/0/2	1/2/2	**2/2/2**	0/0/2	0/1/2
spi	-O0	0/0/5	0/5/5	**7/5/5**	0/0/5	0/5/5
	-O1	0/0/5	0/5/5	**8/5/5**	0/0/5	0/4/5
	-O2	0/0/5	0/5/5	**10/5/5**	0/0/5	0/4/5
	-O3	0/0/5	0/5/5	**10/5/5**	0/0/5	0/4/5
timer	-O0	0/0/1	0/1/1	**1/1/1**	0/0/1	0/1/1
	-O1	0/0/1	0/1/1	**1/1/1**	0/0/1	0/1/1
	-O2	0/0/1	0/1/1	**1/1/1**	0/0/1	0/1/1
	-O3	0/0/1	0/1/1	**1/1/1**	0/0/1	0/1/1
twi_sensor	-O0	0/0/5	0/5/5	**7/5/5**	0/0/5	0/5/5
	-O1	0/0/5	0/5/5	**8/5/5**	0/0/5	0/4/5
	-O2	0/0/5	0/5/5	**10/5/5**	0/0/5	0/4/5
	-O3	0/0/5	0/5/5	**10/5/5**	0/0/5	0/4/5
uart	-O0	1/1/4	0/4/4	**9/4/4**	0/0/4	0/4/4
	-O1	0/0/4	0/4/4	**10/4/4**	0/0/4	0/4/4
	-O2	0/0/4	2/4/4	**12/4/4**	0/0/4	0/2/4
	-O3	0/0/4	2/4/4	**12/4/4**	0/0/4	0/2/4

Table elements: (# of found edges to registered function)/(# of found registered function entry blocks)/(# of known registered functions).

code region, if the jump target has a null value, and if the jump target cannot be disassembled. These all generally correspond to invalid jump targets, which we assume would indicate the end of the table. According to Cojocar's findings, such scenarios are unlikely to come into play, but we feel it appropriate to include them nonetheless. Note however that `switch` statements do not always translate to table branch instructions and FFXE currently does not handle other compilation patterns.

5 Evaluation and Results

5.1 Registered Function Resolution

As we have discussed at length, our algorithm is designed to resolve indirect jumps in firmware whose targets are dependent on volatile memory locations. Thus, our primary interest is whether our technique is able to resolve functions that are called following a registration pattern. To evaluate this, we consider a set of 8 firmware taken from the nRF52 SDK peripheral examples that are known to use the function registration pattern. We compile each sample using the `arm-none-eabi-gcc` [26] compiler at 4 optimization levels in order to maximize the variation in binaries and effectively bring our test set size up to 32. We execute our algorithm, as well as several others, on each of the samples, then determine each algorithm's success at finding known functions based on the presence of both a function's entry block and a valid control flow edge to that block in the graph. The location and presence of each registered function are determined manually by analyzing the source code and unstripped disassembly. The results of our analysis are listed in Table 1, in which each cell lists the number of known registered functions, the number of function entry blocks found, and the number of edges leading to those blocks from right to left. As can be observed, our algorithm can resolve callback function blocks as well as the edges leading to them, giving us confidence in the technique's ability to resolve volatile-influenced indirect branches. Note that we consider registered functions resolved when both the function entry block and an indirect jump to that block are found. However, the table records only the number of edges that lead to functions, without consideration for the control flow transfer type. We have manually analyzed this and discuss the results further in Section 6.

5.2 Overall Coverage Comparisons

Of secondary interest is how our technique compares to other algorithms overall. The goal of CFG recovery is to obtain a high-quality graph that is as close to the ground truth as possible, where the ground truth is represented by the ideal CFG which contains the set of all reachable basic blocks and all realizable control flow transfer edges. CFG quality is therefore judged on 2 primary criteria: *soundness* and *completeness*. We adopt the definitions of soundness and completeness in alignment with the `angr` authors, who flip the definitions of Xu et al. [48]: the recovered CFG G_R is *sound* with respect to the ideal CFG G_I if all edges in the ideal CFG are in the recovered CFG. Conversely, G_R is complete with respect to G_I if all edges in G_R are in G_I. Thus, a perfectly reconstructed CFG is both sound and complete, neither missing vertices nor edges, nor having non-existent ones, while an empty graph would be considered complete, and a fully-connected graph would be considered sound [42].

In an ideal world, we would evaluate the results of our CFG recovery algorithm against a ground truth CFG. However, short of painstaking manual analysis, it is rather difficult to establish the notion of ground truth when considering CFGs recovered from disassembly [35]. A number of works in the literature note that an ideal CFG might be derived directly from source code, but the resulting CFG does not necessarily map well to binary. Moreover, the accuracy of indirect jump edges resolved this way is not always guaranteed [36,48]. The same works opt to approximate the ground truth CFG by taking the union of observed execution paths from a curated set

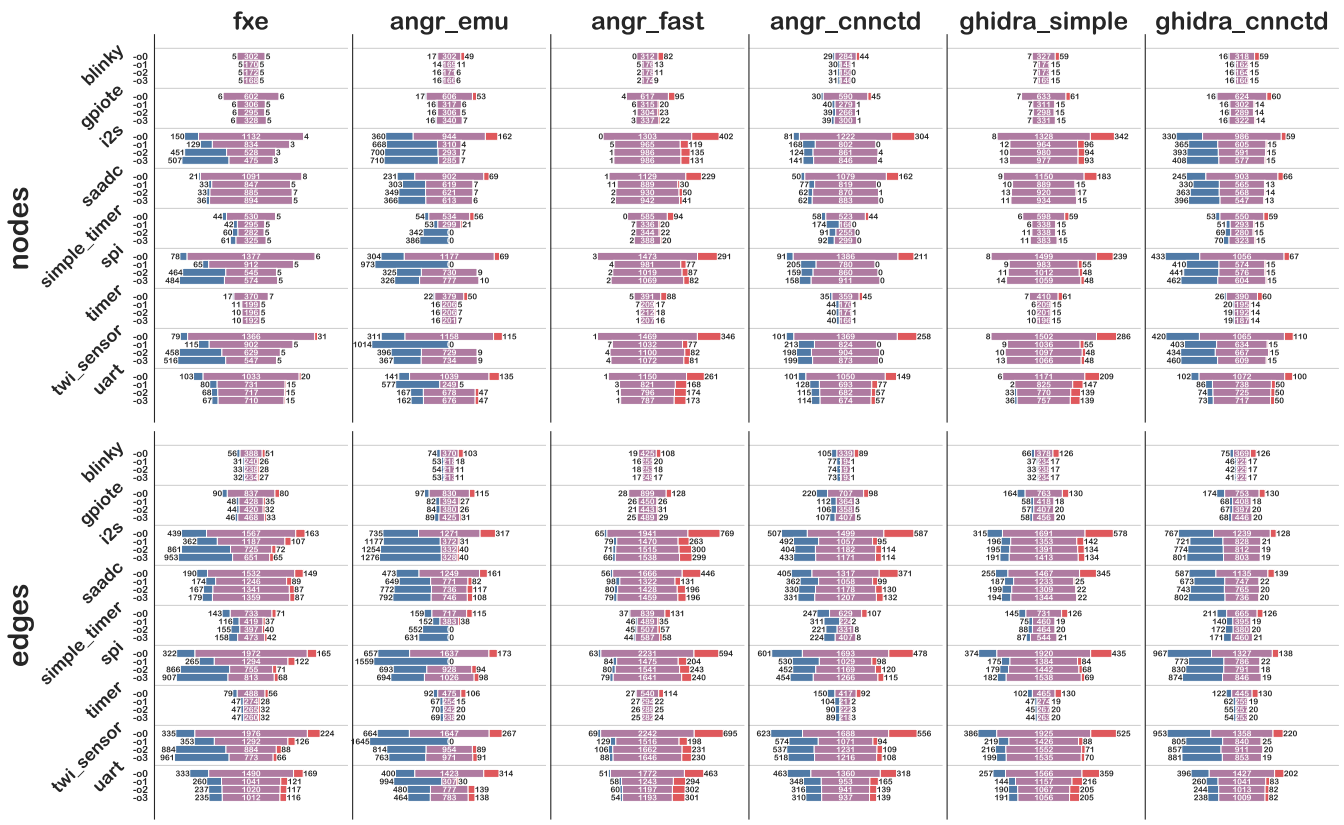

Figure 5: CFG Overlap Between FFXE and other Algorithms (Blue: FFXE only, Red: Column Technique only, Purple: overlap).

of inputs, designed to trigger as many conditions as possible. Unfortunately, designing those inputs for maximum coverage is cumbersome, and while some papers attempt a similar approach via fuzzing [50], the fuzzing of bare-metal firmware is a significantly difficult task. For the same reasons, generally, the union of inputs method is much more difficult to apply to entire firmware images, since firmware inputs typically come from hardware sources and would require significantly more effort to instrument than simply invoking a program with different parameters.

Having established that, short of exacting manual analysis, obtaining an ideal CFG is generally infeasible, we instead adopt the approach taken by other works on CFG recovery [35, 36, 39, 50] and evaluate our method against existing CFG recovery tools by comparing overlap between recovered blocks and edges. In particular, we compare FFXE against our implementation of the original paper's FXE [48], `angr`'s speed-optimized static method and accuracy-optimized dynamic method [42], as well as `Ghidra`'s built-in static CFG recovery technique [31]. We use the same set of firmware binaries compiled in Section 5.1, with the addition of the `blinky` firmware example, for a total of 36 firmware images. The results can be seen in Figure 5, which shows the edge and block overlap between FFXE and the other engines. Specifically, we define an overlapping block as any block in one

Table 2: Real-World Firmware Set

Firmware	Device	Processor	Arch
chargehr_18_32.bin	FitBit Charge HR	STM32L151QD	ARM Cortex-M3
chargehr_18_128.bin	FitBit Charge HR	STM32L151QD	ARM Cortex-M3
flex_7_64_flash.bin	FitBit Flex 1	STM32L151RC	ARM Cortex-M3
flex_7_81_flash.bin	FitBit Flex 1	STM32L151RC	ARM Cortex-M3
flex_7_88_flash.bin	FitBit Flex 1	STM32L151RC	ARM Cortex-M3
polypus_bcm20702a1.bin	Asus USB Dongle	BCM20702A1	ARM7TDMI
switchmate_bright_1_46.bin	Switchmate Bright	nRF52832	ARM Cortex-M4
switchmate_bright_2_9_11.bin	Switchmate Bright	nRF52832	ARM Cortex-M4
switchmate_light_2_21.bin	Switchmate Bright	nRF51822	ARM Cortex-M3
switchmate_light_2_99_16.bin	Switchmate Bright	nRF51822	ARM Cortex-M3

engine whose address range is completely covered by one or more blocks of the other engine. Hence, the blocks found exclusively by one engine in Figure 5 are those blocks that contain addresses not found by the other engine. This is to account for the fact that 2 engines may locate blocks covering the same address range, but with different sizes and boundaries due to differences in edge resolution. Additionally, note that results denoted as `cnnctd` correspond to the *reachable* subgraph of that engine's statically recovered CFG, where only blocks and edges that have a path from a known entry point are kept from the original. We discuss the results in Figure 5 further in Section 6.

In addition to these unit tests, we also run FFXE on a set of real-world firmware samples sourced from 4 commercial

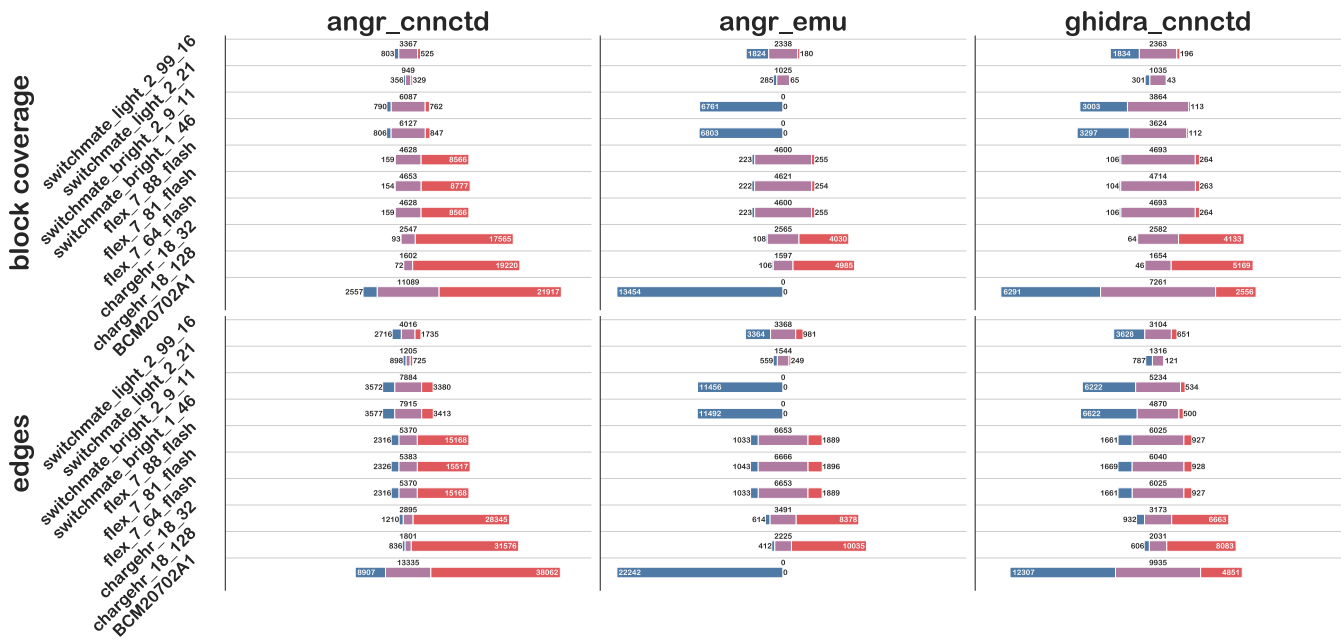

Figure 6: Block Coverage and Edge Overlap Comparison for Real-World Samples (Blue: FFXE only, Red: Column Technique only, Purple: overlap).

products: (1) the FitBit Flex 1 smartwatch, (2) the FitBit Charge HR smartwatch, and (3) the Switchmate Bright and (4) Light smart light switch controllers. The firmware names and processor models can be seen in Table 2, and the CFG coverage comparisons between FFXE and other engines can be seen in Figure 6. Note that in this figure, it is block coverage that is being charted, such the blue bars represent basic blocks found by FFXE that have *no address overlap* with blocks from the other engine. In other words, the exclusive blocks found by either engine would cover address ranges that are not found by other engines. Also note that FFXE is compared to `angr_cnnctd` and `ghidra_cnnctd`, which are the subset of blocks found by the static techniques that are reachable from a known entrypoint.

5.3 Performance Comparisons

While our FFXE implementation is only a prototype and not optimized for execution time, we are nonetheless interested in how its runtime compares to other techniques. As dynamic analysis fundamentally requires longer execution times, for what are likely obvious reasons–an entire CPU must be emulated as opposed to purely examining disassembly–we are inclined to compare FFXE runtime only against that of other dynamic techniques, i.e. the FXE re-implementation and `angr`'s CFGEmulated. To realize this objective, we use the same test set as in Section 5.2. The execution times are listed in Table 3 for the same set of firmware in Section 5.2 and were measured on an Apple M1 Pro processor. We observe that FFXE is considerably faster than `angr`'s emulated method, and somewhat slower than the original implementation. This is about what is expected given the relative complexity of each of the recovery techniques in question. As mentioned before, `angr` has a multi-step recovery process: first lifting the binary to the VEX intermediate representation, running its own symbolic version of dynamic forced execution as a preliminary step, then applying symbolic execution and backward slicing to accurately resolve every encountered indirect branches. Naturally, this translates to longer runtimes, as both symbolic execution and backward slicing involve graph traversal, and symbolic execution must further invoke constraint solving, which has a significant runtime overhead. With respect to the FXE implementation, we find relatively small increases in runtime that likely correspond to the traversal of additional paths. Overall, we are satisfied with the performance of our prototype, as it suggests that the runtime overhead introduced by volatile memory tracking and restoration is relatively modest. Improving the performance of our proposed method is a subject of our future work.

5.4 Case Study: Resolving Complex Data Flows

Our lab collaborates with the health technology startup healthetile.io and they have given their permission for us to conduct a case study on firmware for their We-Be Band, a smartband designed for monitoring vitals like heart rate and blood pressure. They have provided us with a firmware image and graciously allowed us access to their source code so that we can conduct an audit for the presence of vulnerabilities.

Table 3: CFG Recovery Execution Time Comparison.

Firmware		angr_emu	**ffxe**	fxe
blinky	-O0	3.3847	**0.1043**	0.0676
	-O1	1.3367	**0.0752**	0.0508
	-O2	1.3423	**0.0994**	0.0493
	-O3	1.3385	**0.1205**	0.0541
gpiote	-O0	6.5158	**0.7406**	0.1956
	-O1	2.6518	**0.2221**	0.1075
	-O2	2.8654	**0.2520**	0.1254
	-O3	3.0719	**0.3162**	0.2552
i2s	-O0	15.6817	**1.5442**	0.4852
	-O1	2.8067	**0.9330**	0.3883
	-O2	3.0304	**0.9931**	0.2578
	-O3	3.0753	**1.1616**	0.2479
saadc	-O0	16.1505	**1.7066**	0.5687
	-O1	8.3874	**1.1962**	0.4551
	-O2	10.2699	**1.1024**	0.5462
	-O3	8.8527	**1.1639**	0.5672
simple_timer	-O0	6.8036	**1.5731**	0.3361
	-O1	3.1150	**0.5373**	0.2077
	-O2	3.1150	**0.6180**	0.2185
	-O3	3.1150	**0.9832**	0.3324
spi	-O0	20.8080	**3.5622**	1.0024
	-O1	20.8080	**1.5054**	0.6838
	-O2	14.1050	**1.8330**	0.4394
	-O3	12.3269	**2.2592**	0.5342
timer	-O0	4.5828	**0.8804**	0.3094
	-O1	1.5755	**0.4487**	0.1877
	-O2	2.7397	**0.4738**	0.1983
	-O3	2.2008	**0.4726**	0.2006
twi_sensor	-O0	21.5604	**5.8700**	1.2674
	-O1	21.5604	**2.3000**	0.8712
	-O2	12.5984	**2.6013**	0.6310
	-O3	12.0746	**2.5879**	0.5627
uart	-O0	24.9147	**2.5149**	1.0974
	-O1	2.5656	**1.7389**	0.7607
	-O2	11.2223	**1.9881**	0.8162
	-O3	10.4894	**1.9091**	0.7826

Times are recorded in seconds.

While there were a number of potential issues we found in their source code, there was one vulnerability that we found which represents a data-flow path that would otherwise go undetected if only using CFGs recovered with `angr` or Ghidra. Since we were unable to find a plug-and-play automated vulnerability detection tool that employs binary data-flow analysis, we instead identify the data-flow using Ghidra's built-in backward slicing capabilities for decompiled variables and memory referencing capabilities.

This vulnerability can be classified as a potential buffer overflow resulting from a lack of bounds checking for a buffer copy. Because the target buffer is located on the stack, and

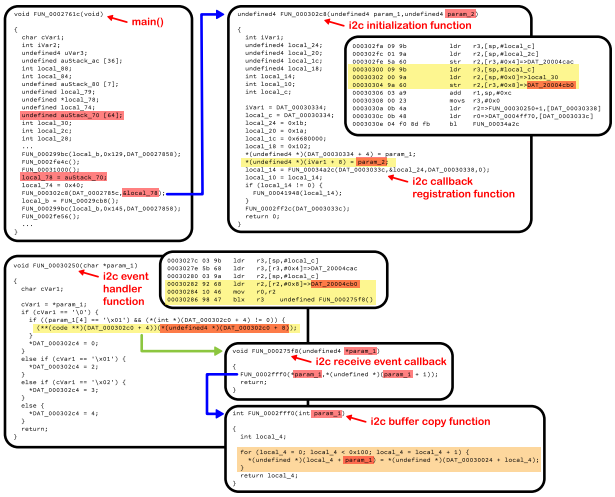

Figure 7: Backward slice of target buffer

the source buffer is read from I2C, the vulnerability is theoretically exploitable. The relevant portions of the backwards slice can be seen in the Figure 7, where we show portions of both the disassembly and decompiled code. We have also verified the decompilation against the source code and confirmed that the decompilation is more or less accurate. From our analysis, it is clear that tracing the affected buffer to assess exploitability was only possible because we were able to resolve calls to registered callback functions using FFXE. The indirect calling edges were not present in static CFGs produced by either Ghidra or angr. Additionally, `angr`'s emulated CFG recovery analysis failed to execute, likely due to lack of implementation details. Yellow highlighted regions map disassembly between decompilation. As can be seen, the first function registers both a callback function, as well as a stack-allocated buffer. In the second set of mapped functions, we can observe the loading of the stack-allocated buffer and its subsequent passing as an argument to an indirect function call, which is presumably a registered callback function. In orange we see the insecure memory copy loop, which copies the entirety of a global I2C buffer to a target buffer without checking the target buffer's length. From the disassembly, we are able to backtrack the target buffer's location to a stack address, passed as a parameter into function FUN_000302c8, which is evidently responsible for registering the callback function to a global structure at 0x20004cb0. Note that the data flow here occurs across threads, since the target buffer is registered in the main thread, but then overwritten in an interrupt context. Because neither Ghidra nor `angr` were able to recover the control flow paths to the callback functions, which is where the overflow occurs, they are unable to trace the target buffer all the way back to its source, which is necessary to determine if the vulnerability is technically exploitable. The presence of the unsafe buffer write alone is not proof that it can be exploited via external input. We have notified our collaborator of the vulnerability and have been informed that it is

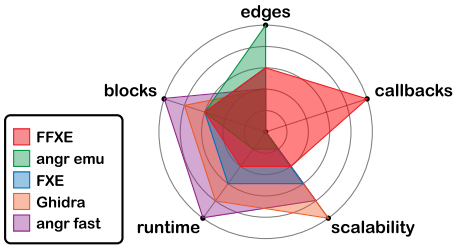

Figure 8: Qualitative Comparison of CFG Recovery Methods.

indeed a bug present in some of the debugging infrastructure that they have implemented. They've informed us that it will be removed. Note that we do not demonstrate the tangible security benefits of FFXE-enhanced CFGs beyond the case study we have presented here, as our focus is primarily on the problem of CFG recovery. This case study is meant to be a proof-of-concept for how FFXE can assist in further analysis.

6 Discussion

This section discusses the results of our evaluation, comparing FFXE to other CFG recovery tools based on our observations. A qualitative comparison of each technique can be found in Figure 8, which ranks orders each technique into 5 categories based on our results and experiences using each tool. Note that in this section, we do not discuss each tool in-depth, but focus on how each tool compares to FFXE.

6.1 Analyzing Identified Registered Function

As mentioned in Section 5.1, we consider a registered function successfully resolved when both the function's entry block, (i.e. the first basic block in the function) and at least one valid edge leading to that block is resolved. Moreover, we have stated that the tabulated edge values had not been validated as actual control flow transfers. Therefore, we must manually analyze each of the cases where blocks and edges are reported found to confirm their validity. We note first that in all test cases for the FFXE, we have fulfilled these conditions and confirmed that each edge does indeed correspond to a valid indirect branch. We can therefore have some confidence in our technique's ability to resolve indirect branches whose targets are written asynchronously to a volatile location. Conversely, we have found that all edges found in other algorithms are fall-through edges, which do not correspond to real control flow transfers. This means that, in general, the entry blocks resolved by the static algorithms–`angr`'s fast method and `Ghidra`'s built-in method–are the result of pattern matching against common function prologues. Hence, in reality, none of the algorithms we compare against are actually capable of resolving volatile memory-dependent indirect branches.

6.2 Considerations for Coverage Comparison

6.2.1 Comparison with Dynamic Techniques

Our coverage results are shown in Figure 5. We can observe improvement over the other dynamic methods, FXE and `angr`'s emulated algorithm, evidenced by the significant number of blocks and edges that were found exclusively by FFXE, as opposed to those found exclusively by the other methods. Furthermore, we demonstrate improvement over the original FXE implementation, in that our recovered CFGs essentially subsume those found by the original, finding nearly all blocks and edges that the original finds, and additionally those part of registered callback functions. This is expected because our modifications to the original algorithm are relatively minor, and do not alter the core mechanisms involved. We are merely adding some functionality to account for asynchronicity. Because the majority of the original algorithm remains intact, we are able to resolve a CFG that is a superset of the original. Hence we can claim that when compared to the original, our enhanced version is able to provide a more sound graph, though this necessarily increases uncertainty about its completeness.

In comparison with `angr`'s emulated recovery method, we observe significantly more coverage, which leads us to claim greater soundness for our recovery technique. We further investigate the underlying causes for the results by conducting an automated analysis on the blocks and edges exclusively found by `angr`. Due to space limitations, the tabulation of our analysis can be found on our Github. [2]

We observe that `angr`'s CFGEmulated algorithm prioritizes accuracy and completeness over coverage, and stores a significant amount of runtime information to enable further analyses like backward slicing. In order to optimize for accuracy and completeness, it makes significant use of value-set analysis as well as constraint solving and symbolic execution when encountering indirect jumps. Because it takes this approach, nearly all basic blocks that it finds are valid, and tends to include all technically possible indirect jumps. The `angr` documentation notes that indirect jumps resolved this way are not guaranteed to be always taken. In our investigation of recovered basic blocks we found an average error rate of approximately 59%, in which the invalid blocks were erroneously disassembled data regions, or corresponded to dead `nop` instructions. Likewise, when considering recovered edges, we found an average rate of error of approximately 19%, in which the edge started at or led to a false address. When we consider the total number of blocks and edges resolved, FFXE almost always outperforms `angr`'s dynamic recovery. Aside from FFXE's ability to resolve registered callback functions, we believe the primary reason for the difference is that FFXE employs concrete execution, while `angr` employs symbolic execution. As a result, `angr` must always

[2] https://github.com/rchtsang/ffxe/tests/analysis

rely on constraint solving, value set analysis, or some other additional method of indirect jump resolution when an indirect jump is encountered, as opposed to having a concrete target by default as FFXE does. While this does afford `angr` some advantages, this puts much more burden on the indirect jump resolution techniques for successful target resolution. If resolution fails, further execution cannot continue down that path. By contrast, because FFXE defaults to a concrete value, it is able to continue exploring. The additional cost also shows up in the form of additional execution time, since relying on constraint solvers for every indirect jump encountered is expensive. For this reason, `angr` also limits the call depth for function re-execution to 1. Our understanding of this is that, after a function has been explored once, if is seen again, it will only execute if calling that function will not increase the depth of the call stack past the limit, further limiting the potential for block resolution.

6.2.2 Comparison with Static Techniques

Comparison against static recovery techniques is not as straightforward as against other dynamic techniques, due to the fact that the underlying mechanisms are significantly different. As we have already discussed in earlier sections, static methods often use pattern matching and fall-through edges for the recognition of basic blocks and are limited in their ability to resolve indirect jumps. This tends to lead to higher block coverage, but diminished completeness, as fall-through edges do not always correspond to valid control flow transfers. We believe that these differences give a partial account for the relative lack of overlap between CFGs resolved by FFXE and those by static methods that appear to be present for both `angr`'s fast static recovery (`angr_fast`) and `Ghidra`'s default recovery (`ghidra_simple`), encoded in the size of the red bars charted in Figure 5.

To understand the differences in resolved blocks and edges between FFXE and static techniques, we have analyzed the blocks and edges found exclusively by `angr` and Ghidra for our unit tests on the nRF52832, since these binaries have corresponding source code and unstripped elf files whose disassembly can act as ground truth in determining whether a block is valid. This allowed us to programmatically determine if a recovered block represented a real block, or was erroneously disassembled data. Likewise for edges, we were also able to filter edges that led to falsely disassembled instructions in data sections or between actual instructions. As expected, for `angr_fast` and `ghidra_simple`, there were significantly more blocks and edges found than FFXE, since static pattern matching strategies are not constrained by actual control flow and, as a result, can pick up dead code or falsely disassemble data. On average, about 62% and 48% of statically recovered blocks, for `angr` and Ghidra respectively, corresponded to erroneously disassembled addresses, and about 32% and 41% of edges either started at or led to a false address, making

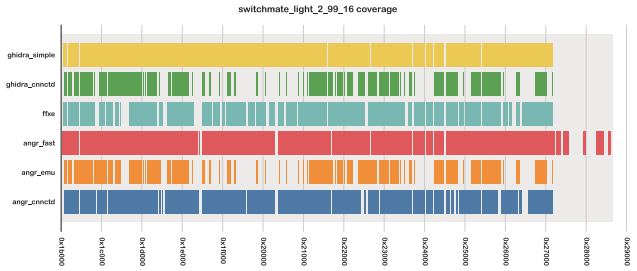

Figure 9: Visualization of coverage across the firmware image

them invalid. As we hoped, this is in line with our goal of producing more accurate CFGs, particularly with regard to completeness.

Additionally, we conducted a random sampling of the statically recovered blocks and edges that were not marked invalid, then performed a manual analysis to identify the reasons why FFXE could not resolve them. We determined that most of the blocks in this category had one of 2 causes for lack of identification: the blocks were part of unreachable dead code regions (about 60% and 59% for `angr` and Ghidra respectively), or the blocks could be traced back to a switch statement compiled as a load-based jump table (35% and 33%), which FFXE does not handle. We also note that the compiler optimization significantly influences this latter statistic, as unoptimized code tends to include many more functions and code sections that would otherwise be optimized away, including the aforementioned load-based jump tables. This appears to account for much of the differences in block resolution between optimized and unoptimized firmware images. As for edges, load-based branch tables (28% `angr` and 44% Ghidra) and dead code (21% and 49%) again played a significant role, with the difference that `angr` appeared to identify more function return edges (39%).

Based on our in-depth analysis of those blocks and edges not found by FFXE, we believe that FFXE provides complementary coverage gains to existing techniques, where existing techniques have a definite edge in resolving load-based branch tables, but fall short in completeness. Indeed, when we strip away the dead code regions in the `cnnctd` graphs, Figure 5 suggests FFXE can markedly improve completeness with the additional resolution of indirect branches. The complete tabulation of our manual analysis can also be found on our Github.

6.3 Effectiveness on Real-World Firmware

The high-level coverage comparisons between FFXE and every other algorithms are included on our Github. As already mentioned, in general, static recovery techniques tend to outperform dynamic techniques in terms of pure coverage. However, when we consider CFG coverage in terms of blocks that are reachable from a known entrypoint, for 8 out

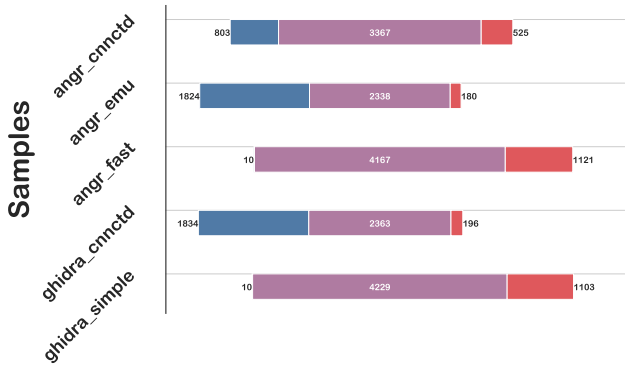

Figure 10: Basic block coverage overlap

of the 10 firmware images, FFXE coverage is comparable or substantially complementary to other techniques. In all cases of coverage comparison, we found that FFXE reports blocks that have non-overlapping coverage with other tools, which implies that FFXE has complementary utility with existing techniques. Note this is especially true of the images for the Switchmate Light and Bright, one sample of which is shown in Figures 9 and 10. Again, note that when compared to static tools, FFXE is helpful for recovering reachable blocks as opposed to sheer coverage of the address space. In the example above, this can be seen when comparing the overlaps to connected ("cnnctd") graphs, which are simply the results of doing a full traversal of the statically recovered CFG from any known entry point. While nearly all of FFXE's blocks are covered by static analysis, when the static CFG (whether from Ghidra or from `angr`) is reduced to only what is reachable from entry points, the coverage is significantly diminished. This is because both `angr` and Ghidra apply a mixture of pattern matching and recursive disassembly when conducting their analysis, which allows them to aggressively identify potential functions for disassembly, but cannot provide the control flow edges that would lead to them. This, as mentioned earlier, can impede automated analyses. Hence, our tests on real-world firmware demonstrate that the complimentary control flow edges that FFXE resolves can benefit analysis algorithms that rely on accurate representations of control flow between blocks.

6.4 Shortcomings

Here we must clarify what we believe to be the current shortcomings of FFXE:

- ***Increased Memory Footprint:*** To restore context upon asynchronous writes to volatile memory, we employ the simple but memory-intensive strategy of making additional backups of the processor and memory state. As this is already a costly task, we potentially exacerbate the problem by keeping these additional backups.
- ***Exploration Cycle Uncertainty:*** In the original forced execution paper, the quota system is introduced to guarantee termination of the depth-first search, and optimizations are made to ensure that enough state information is captured to lead to valid indirect jumps. We leave this system intact in our implementation and do not account for all edge cases that may arise when restoring a state based on a volatile memory write. While we believe that the quota system should handle this without modification, we have conducted no rigorous proof of this claim, and therefore admit uncertainty in the occurrence of infinite loops that may result from improper handling of block quotas in conjunction with cycles in the CFG.
- ***Invalid Processor States:*** The original paper was based on the assumption that indirect branch targets were generally independent of intermediate states. This assumption has been found to be false on multiple occasions [36,42]. While our work discards this assumption implicitly, as the very notion of volatile memory states must necessarily invalidate the assumption, the rest of the algorithm has not been touched, and may therefore still be subject to the shortcomings that accompany the invalid processor states when conditional statements are ignored.
- ***Architecture Dependence:*** Our prototype is currently coupled to a particular hardware platform and ISA. While we have created it with some flexibility and the ability to adapt to other hardware platforms that implement the same ISA, it is still fundamentally tied to the underlying architecture, particularly the Cortex-M interrupt management system. This means that, while not impossible, some implementation effort is necessary to apply this technique to other architectures.
- ***Scalability:*** Given our observations of execution time compared against both angr and the original FXE implementation, the scalability of our technique is not entirely certain. By adding additional context backups and path explorations on top of those needed by the original algorithm, we introduce some overhead in both execution time and memory footprint. However, it is difficult to predict how this overhead scales with firmware size without further analysis.

7 Future Work

Based on our observations in Section 6, we believe there are several practical directions that this research might take. First, because we have observed that our modification to the original forced execution algorithm adds only modest overhead to execution time, we believe that the general concept of volatile-aware forced execution might be incorporated into more mature existing dynamic CFG recovery techniques, like that of `angr`'s emulated method. Doing so could make `angr`'s

already powerful recovery technique much more suitable for firmware analysis. Furthermore, doing so would directly address the aforementioned issue of invalid processor states, as `angr`'s emulated method directly addresses this shortcoming through the use of symbolic execution. Conversely, we can consider enhancing FFXE further by adapting some of `angr`'s techniques and attempting to address the issues of invalid processor states, architecture dependence, and lack of features using intermediate representations and symbolic execution.

Second, we consider further evaluation of our technique by expanding our test set with more real-world examples and comparing against more recovery methods, such as those in IDA Pro [17], CMU's Binary Analysis Platform [9], and other research works like X-Force [36]. Expanding both our test set and comparison space would give us greater confidence in the validity and effectiveness of our technique.

Third, we consider possible enhancements to improve scalability, by addressing performance overheads in execution time and memory. Regarding memory footprint, we can refactor our codebase to utilize a database approach to saving and restoring execution contexts, allowing us to eliminate memory overhead that may arise from unnecessary duplication during backup and restore steps. Regarding execution time, if the quota system can be rigorously re-examined, we may be able to minimize the number of search paths for indirect branches and reduce the total execution time. Additionally, because forced execution is fundamentally a graph traversal, we may also be able to introduce speedup by designing a parallelized version of the algorithm.

Finally, to build upon this work and truly examine its potential impact on broader-scale firmware analysis, we intend to enhance the existing prototype to produce graphs that can be incorporated into more sophisticated techniques, such as binary similarity analysis or data flow analysis. Examining and comparing the results of such techniques when using CFGs generated using various recovery methods can give insight into how firmware analysis might benefit from enhancing intermediate analyses like control flow graph recovery with hardware awareness.

8 Conclusion

In this paper, we have presented FFXE, our prototype for a modified form of forced execution designed to address indirect branches whose targets are dependent on volatile memory by saving and restoring execution contexts on volatile memory accesses. We have tested our prototype on a test set of 36 example images, 32 of which contain manually verified functions that rely on the callback registration pattern. We compared our results against a re-implementation of the original forced execution algorithm, as well as the default CFG recovery methods employed by the popular software analysis frameworks `angr` and `Ghidra`. We find that in all 32 cases, we are able to meaningfully locate the registered callback functions where other techniques fail to do so. This gives us confidence that our implementation of volatile-aware forced execution is a viable solution to this class of indirection, which appears commonly in embedded firmware binaries. We have found that our tool has potential for improvement, but that the underlying algorithm is conceptually simple enough that other tools may benefit from incorporating this technique into their own dynamic recovery algorithms. We hope that our work will enable the application of sophisticated software analysis techniques to firmware and ultimately lead to more secure embedded systems.

Acknowledgments

This work was funded in part by NSF CHEST industrial sponsors and the Robert N. Noyce Trust.

Availability

Our code artifacts are made publicly available at https://github.com/rchtsang/ffxe

References

[1] Shahid Alam, Issa Traore, and Ibrahim Sogukpinar. Annotated control flow graph for metamorphic malware detection. *The Computer Journal*, 58(10):2608–2621, October 2015.

[2] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, March 1976.

[3] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, July 1970.

[4] ARM Ltd. Nested vectored interrupt controller. Online. accessed 6-February-2023.

[5] Thomas Baar and Sergey Staroletov. A control flow graph based approach to make the verification of cyber-physical systems using keymaera easier. *Modeling and Analysis of Information Systems*, 25(5):465–480, October 2018.

[6] Gogul Balakrishnan and Thomas Reps. Analyzing Memory Accesses in x86 Executables. In Evelyn Duesterwald, editor, *Compiler Construction*, Lecture Notes in Computer Science, pages 5–23, Berlin, Heidelberg, 2004. Springer.

[7] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):1–84, 2010.

[8] Frédéric Besson, Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9(3):217–250, July 2001.

[9] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. *BAP: A Binary Analysis Platform*, volume 6806 of *Lecture Notes in Computer Science*, page 463–469. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[10] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *Proceedings of the Third DIMVA*, DIMVA'06, pages 129–143, Berlin, Heidelberg, July 2006. Springer-Verlag.

[11] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song. Hi-cfg: Construction by binary analysis and application to attack polymorphism. In *Computer Security – ESORICS 2013*. Springer, 2013.

[12] Lucian Cojocar, Taddeus Kroes, and Herbert Bos. JTR: A Binary Solution for Switch-Case Recovery. In *Engineering Secure Software and Systems*, Lecture Notes in Computer Science, pages 177–195, Cham, 2017. Springer International Publishing.

[13] Hoang-Vu Dang and Anh-Quynh Nguyen. Unicorn: Next generation cpu emulator framework, August 2015.

[14] Bjorn De Sutter, Bruno De Bus, Koenraad De Bosschere, Peter Keyngnaert, and Bart Demoen. On the static analysis of indirect control transfers in binaries. In *Proceedings of PDPTA 2000*, pages 1013–1019, Las Vegas NV USA, June 2000. CSREA Press.

[15] Jesse Deveza, Lanier Santos, and Rosiane de Freitas. Control flow graph, formal verification and constraint programming techniques. In *10th LAWCG*, June 2022.

[16] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.ng: A unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 131–141, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Hex-rays. Ida pro. Online. accessed 6-February-2023.

[18] Marieke Huisman and Dilian Gurov. Cvpp: A tool set for compositional verification of control–flow safety properties. In *Formal Verification of Object-Oriented Software*, Lecture Notes in Computer Science, pages 107–121, Berlin, Heidelberg, 2011. Springer.

[19] Giacomo Iadarola, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. Call graph and model checking for fine-grained android malicious behaviour detection. *Applied Sciences*, 10(22):7975, January 2020. Number: 22 Publisher: Multidisciplinary Digital Publishing Institute.

[20] YU-liang LU Kai-long ZHU. Construction approach for control flow graph from binaries using hybrid analysis. *Journal of ZheJiang University (Engineering Science)*, 53(5):829–836, May 2019.

[21] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. Refining indirect call targets at the binary level. In *NDSS*, 2021.

[22] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 214–228, Berlin, Heidelberg, 2009. Springer.

[23] Tim Lange, Martin R. Neuhauber, and Thomas Noll. Ic3 software model checking on control flow automata. In *FMCAD*, pages 97–104, Austin, TX, USA, September 2015. IEEE.

[24] ARM Ltd. *DUI0553B Cortex-M4 Devices Generic User Guide*. ARM Ltd., 2011.

[25] ARM Ltd. *DDI0403E ARMv7-M Architecture Reference Manual*. ARM Ltd., 2014.

[26] ARM Ltd. Arm gnu toolchain, 2023. Online. accessed 2-February-2023.

[27] Yi-Fan Ma and Ming Li. The flowing nature matters: feature learning from the control flow graph of source code for bug localization. *Machine Learning*, 111(3):853–870, March 2022.

[28] Zhuo Ma, Haoran Ge, Yang Liu, Meng Zhao, and Jianfeng Ma. A combination method for android malware detection based on control flow graphs and machine learning algorithms. *IEEE Access*, 7:21235–21245, 2019. Conference Name: IEEE Access.

[29] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 24–35, New York, NY, USA, July 2016. Association for Computing Machinery.

[30] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 231–245, May 2007. ISSN: 2375-1207.

[31] National Security Agency. Ghidra software reverse engineering framework, 2023. Online. accessed 3-February-2023.

[32] Minh Hai Nguyen, Thien Binh Nguyen, Thanh Tho Quan, and Mizuhito Ogawa. A hybrid approach for control flow graph construction from binary code. In *20th APSEC*, volume 2, pages 159–164, December 2013. ISSN: 1530-1362.

[33] Coseinc Nguyen Anh Quynh. Capstone: Next-gen disassembly framework, 2014. Online. accessed 6-February-2023.

[34] Nordic Semiconductor. nrf5 sdk: Software development kit for the nrf52 series and nrf51 series socs, 2023. Online. accessed 26-January-2023.

[35] Chengbin Pang, Tiantai Zhang, Ruotong Yu, Bing Mao, and Jun Xu. Ground truth for binary disassembly is not easy. In *31st USENIX Security Symposium*, pages 2479–2495, Boston, MA, August 2022. USENIX Association.

[36] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-force: Force-executing binary programs for security applications. In *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, August 2014. USENIX Association.

[37] Thomas Reinbacher and Jörg Brauer. Precise control flow reconstruction using boolean logic. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 117–126, New York, NY, USA, October 2011. Association for Computing Machinery.

[38] Andrei Rimsa, José Nelson Amaral, and Fernando Magno Quintão. Efficient and precise dynamic construction of control flow graphs. In *Proceedings of the 23rd SBLP*, SBLP '19, pages 19–26, New York, NY, USA, September 2019. Association for Computing Machinery.

[39] Andrei Rimsa, José Nelson Amaral, and Fernando M. Q. Pereira. Practical dynamic reconstruction of control flow graphs. *Software: Practice and Experience*, 51(2):353–384, February 2021.

[40] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering*, pages 45–54. IEEE, 2002.

[41] Nordic Semiconductor. *nRF52832 Product Specification*. Nordic Semiconductor, 2017.

[42] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[43] Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming Languages*, 3(POPL):48:1–48:29, January 2019.

[44] Henrik Theiling. Extracting safe and precise control flow from binaries. In *Proceedings Seventh RTCSA*, pages 23–30, December 2000. ISSN: 1530-1427.

[45] Liam Tung. Microsoft: Firmware attacks are on the rise and you aren't worrying about them enough. *ZDNet*, 2021.

[46] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 167–180, Virtual Event USA, October 2020. ACM.

[47] Na Xiao, Jing Zeng, Qigui Yao, and Xiuli Huang. A method of firmware vulnerability mining and verification based on code property graph. In *Advances in Artificial Intelligence and Security*, Communications in Computer and Information Science, pages 543–556, Cham, 2022. Springer International Publishing.

[48] Liang Xu, Fangqi Sun, and Zhendong Su. Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep*, pages 14–23, 2009.

[49] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. Pmp: Cost-effective forced execution with probabilistic memory pre-planning. In *IEEE Symposium on Security and Privacy (SP)*, pages 1121–1138. IEEE, 2020.

[50] Kailong Zhu, Yuliang Lu, Hui Huang, Lu Yu, and Jiazhen Zhao. Constructing more complete control flow graphs utilizing directed gray-box fuzzing. *Applied Sciences*, 11(3):1351, January 2021. Number: 3 Publisher: Multidisciplinary Digital Publishing Institute.

Appendix A Tabulated Data and Additional Figures

Note that additional tables and visualizations can be found on our Github at https://github.com/rchtsang/ffxe/tests/analysis.

Table 4: Real-World Test Results

Firmware		angr_cnnctd	angr_emu	angr_fast	**ffxe**	ghidra_cnnctd	ghidra_simple
BCM20702A1	blocks	32805	0	44932	**13454**	9682	37148
	edges	51397	0	79441	**22242**	14786	57788
chargehr_18_128	blocks	20778	6479	23763	**1640**	6764	22561
	edges	33377	12260	44102	**2637**	10114	35169
chargehr_18_32	blocks	20040	6454	23140	**2574**	6629	21556
	edges	31240	11869	43202	**4105**	9836	33370
flex_7_64_flash	blocks	13061	4656	18881	**4698**	4783	14162
	edges	20538	8542	31230	**7686**	6952	21678
flex_7_81_flash	blocks	13286	4666	19755	**4712**	4793	14210
	edges	20900	8562	32418	**7709**	6968	21741
flex_7_88_flash	blocks	13061	4656	18881	**4698**	4783	14162
	edges	20538	8542	31230	**7686**	6952	21678
switchmate_bright_1_46	blocks	6679	0	7792	**6803**	3492	7599
	edges	11328	0	14521	**11492**	5370	12013
switchmate_bright_2_9_11	blocks	6610	0	7756	**6761**	3756	7455
	edges	11264	0	14703	**11456**	5768	11778
switchmate_light_2_21	blocks	1254	1072	1857	**1299**	1001	1673
	edges	1930	1793	3201	**2103**	1437	2439
switchmate_light_2_99_16	blocks	3823	2459	5206	**4129**	2436	5148
	edges	5751	4349	9077	**6732**	3755	8007