ChatGPT

Software Obfuscation in 8032/8051 Firmware and Reverse-Engineering Techniques

Introduction: The Intel 8051 (MCS-51) and its derivatives (often called 8032/8052 for enhanced versions) remain widely used in embedded devices – including USB controllers like the SN9B292B – decades after their introduction [1] [2]. To protect proprietary firmware and deter attacks, vendors increasingly employ software-level obfuscation methods. These techniques make the code harder to analyze by humans or tools, even if an attacker manages to dump the firmware (bypassing hardware lock bits or fuse protections) [3]. In the case of USB microcontrollers (e.g. in BadUSB-style attacks), such obfuscation aims to conceal device functionality or malicious logic from security researchers [4] [5]. This report surveys known and emerging obfuscation techniques for 8051-family firmware and how reverse engineers counter them. We focus on strategies relevant to SN9B292B – an 8032-based USB controller – including control-flow tricks, data masking, indirect addressing, mixed binary/assembly code, instruction substitutions, jump manipulations, and encoded constants. We then outline reverse-engineering tools and methods (symbolic execution, IR lifting, static/dynamic analysis, fuzzing, pattern recognition, patching) that analysts use to unravel such obfuscation.

Obfuscation Techniques for 8051/8032 Firmware

Control-Flow Obfuscation: Obfuscators often alter the program's control flow to confuse disassemblers and decompilers. One common method is *control-flow flattening*, which removes the natural structured flow of code and instead routes most behavior through a single dispatcher loop with many case branches [6]. This makes the firmware's call graph and loops appear as spaghetti logic. Related tactics include inserting *opaque predicates* – conditional checks whose outcome is fixed (always true/false) but computed in a complex way – to create fake branches and dead code paths [7] [8]. Attackers also add *dummy jumps* or *irreducible loops* that never execute meaningfully but break analysis tools. For example, an obfuscated 8051 program might jump into the *middle* of an instruction or to an odd address to misalign the code parsing. Such **unaligned jump targets** are highly unusual in normal code; indeed, a statistical scan of 8051 firmware treats any relative jump landing not at an instruction start as suspicious [9]. This could indicate deliberately obfuscated control flow (e.g. jumping into an instruction to reinterpret it differently). Reverse engineers must detect and realign these jumps during analysis. Another trick is **jump table indirection** or **call trampolines**, where legitimate calls are replaced by a lookup in a table of addresses or by pushing a target address to the stack and using `RET` to "jump". These indirections hide the true control-flow targets from naive static analysis. In practice, firmware like the SN9B292B's might use *jump redirection* stubs for important routines – e.g. an interrupt vector that immediately jumps to a computed address – making it harder to follow code in tools.

Data Masking and Constant Encoding: Obfuscation is not limited to code flow; sensitive data and constants can be hidden via arithmetic or encoding. Rather than referencing a constant value directly (which would be easy to spot in a disassembly), the firmware may **embed constants in an obfuscated form**. For example, a numeric constant or key might be split into several bytes that are combined at runtime, or stored XORed with a secret mask. An obfuscator might replace a simple load like `MOV A, #0x5A` with a sequence that computes 0x5A dynamically (e.g. XORing multiple values or using a math formula). One documented approach is to move constants into a lookup table that is then accessed via masked indices [10] [11]. In the context of Java bytecode obfuscation, for instance, constants can be relocated to a hidden array and XOR-masked, with the code doing a lookup and XOR at runtime [10] [11].

The same principle applies to firmware: a routine could retrieve an encoded constant from code memory and decode it on the fly. This defeats simple pattern searches for the constant's bytes in the binary. **Data masking** might also involve constantly XORing or incrementing variables so that their true value is never held in registers for long. These methods increase the reverse engineer's workload, since one must track how each value is derived. If done cleverly, the mask values themselves are also hidden or dynamically generated [10].

Indirect Memory Addressing: The 8051 architecture allows both direct and indirect memory access (e.g. using register `R0`/`R1` as pointers into internal RAM, or the 16-bit `DPTR` register for external memory/code). Firmware obfuscation often leans on **indirect addressing** to obscure which variables or data are being used. For example, instead of `MOV A, 0x30` (directly reading a known I/O register or RAM address 0x30), a program can set `R0` to 0x30 and do `MOV A, @R0`. The effect is the same, but a static disassembler can no longer tell at compile-time which address is being accessed – it sees only the indirect reference. This is particularly prevalent in some USB controller firmwares. Research on two USB 8051 firmwares found that one (Phison controller) "uses direct addressing for most memory reads and condition variables," whereas another (EzHID firmware) "uses **indirect reads from memory for conditional variables**" [12]. In other words, the second firmware likely loads a condition flag via a pointer, making it harder to discern the purpose of that flag without executing the code. Indirect addressing also shows up in control-flow obfuscation: an obfuscated firmware might compute a function pointer in RAM and then call it, rather than calling subroutines directly. Reverse engineers counter this by *resolving pointers* through careful analysis or by instrumented execution (to log what addresses are actually accessed at runtime).

Instruction Substitution and Equivalents: Another layer of obfuscation is replacing common instructions or idioms with less obvious ones. The 8051 has a relatively small instruction set (about 44 mnemonics, 256 opcodes [13]), but there are often multiple ways to do the same thing. For instance, clearing a register can be done with `CLR A` or by `MOV A,#0`. An obfuscator might consistently use a less typical sequence (perhaps `XOR A,A` to clear `A`, or even something like `SUBB A,A`). Arithmetic operations might be replaced with logically equivalent sequences: e.g. instead of `INC X` (increment), use `ADD X,#1`; instead of a straightforward multiply instruction, use a loop of additions. The idea is to break the signatures that tools or experts look for. In high-level code, developers sometimes deliberately write confusing constructs (like computing a value via bit rotations and XORs as in a twisted checksum routine [14]) to prevent easy understanding. On 8051 firmware, compiled C code tends to have recognizable patterns (for loops, switch-case jump tables, standard library calls, etc.), and instruction substitution can disrupt these patterns. **Binary/Assembly hybrid blending** is common in this family: developers may write critical sections in assembly for size/speed, or to implement obfuscation that a C compiler wouldn't produce. These hand-coded assembly snippets can do things like manipulate the program status word (PSW) or stack in unusual ways. For example, an obfuscated routine might manually save and alter the return address on the stack to implement a non-standard jump (this is a form of *stack pivot* or trampolining that high-level code wouldn't show). Such blending of compiled and handcrafted code means the firmware doesn't follow the usual C function structure, confusing decompilers. Indeed, a reverse-engineering blog on 8051 notes that decompilers can misidentify instruction purposes until special SFR (special function register) names and patterns are corrected [15] [16]. This underscores how custom assembly can defy the assumptions of automated analysis.

Jump Redirection and Opaque Jumps: As a specific facet of control-flow obfuscation, many 8051 protections implement **jump misdirection**. The firmware might insert dummy jumps or route legitimate jumps through an extra layer. One approach is to replace a direct call to a function with a jump to a small trampoline that then jumps to the real function. This adds no real runtime benefit, but it means a static analysis tool might not easily realize that the trampoline and target are logically connected. Another trick: after an obfuscated conditional, instead of jumping directly to the "true" or "false" handler, the code might jump into the middle of an unrelated instruction which, when executed from that point, actually performs the desired logic. This exploits the variable-length nature of 8051 instructions. For example, a two-byte instruction in memory, when entered at the second byte, could act as a 1-byte instruction of a different kind. This *leverages overlapping instructions* to create alternate instruction streams. The result is highly opaque to a disassembler, which normally assumes aligned instruction starts. As noted earlier, a perfectly legitimate 8051 program should have **0% of relative jumps targeting non-instruction-starts**, so any non-zero rate is suspect (small deviations can occur for jump tables, but values $\gg 0$ are a red flag) [9]. Reverse engineers must manually adjust the disassembler or use brute-force code emulation to discover the real instruction boundaries in such cases.

Emerging & Advanced Techniques: Beyond these "classical" methods, recent research has proposed more **sophisticated obfuscation strategies for firmware**. One academic example is *runtime code reordering*: a 2019 work introduced a "firmware obfuscation by swapping a subset of instructions" such that the code in memory is scrambled and inoperable until the device bootloader dynamically reconstructs the correct order using a secret key [17] [18]. In that scheme, the mapping of swapped instruction addresses is derived from a device-specific Physically Unclonable Function (PUF), meaning each device's firmware is uniquely arranged [19]. An attacker who reads out the flash contents would see a jumbled instruction sequence and *not know which pairs were swapped* without the key. This approach doesn't encrypt the entire firmware but achieves a similar effect by requiring a hardware-dependent de-obfuscation at runtime. Other state-of-the-art methods incorporate cryptography or hardware support: e.g. one might encrypt certain code segments and decrypt on the fly in microcontroller RAM (the 8051 is resource-limited, but some modern derivatives or external logic could enable this). Researchers have also explored using **Oblivious RAM (ORAM)** techniques to obfuscate memory access patterns, thereby hiding the control flow. Goldreich and Ostrovsky's ORAM (originally a theory for hiding memory access in software) has been suggested to "obfuscate control flow" by ensuring memory fetch patterns do not reveal actual code paths [20]. However, ORAM is heavy for small MCUs. Simpler techniques like **multi-variant firmware** are also discussed: each device could have the same functionality but with different syntactic forms (different instruction ordering, varying register usage, etc.), thwarting mass pattern-matching attacks [21]. In practice, many of these advanced methods are not yet ubiquitous in industry due to cost and complexity. Industry tends to rely on a combination of the earlier techniques (flow flattening, junk code, encoded constants) plus hardware lock bits. Still, as IoT firmware security grows in importance, we see a convergence of academic ideas and industry implementations to protect firmware IP and integrity.

Summary of Firmware Obfuscation vs. Reversing Countermeasures

To summarize the obfuscation techniques and how reverse engineers tackle them, the table below outlines key methods and their counter-strategies:

Obfuscation Technique	How it Protects Firmware	Reverse-Engineering Approach
Control-Flow Obfuscation (flattening, opaque predicates, irreducible loops)	Restructures code into unnatural flows; hides logical branch structure. Example: single loop dispatching many code blocks out of order.	Reconstruct the control-flow graph via static analysis and manual reasoning. Identify dispatcher loops and restore structured logic. Use deobfuscation tools or scripts to detect flattened state machines [6].

Obfuscation Technique	How it Protects Firmware	Reverse-Engineering Approach
Jump Redirection & Unaligned Jumps	Uses trampolines or jumps into middle of instructions to confuse disassembly. Obscures true jump targets and function boundaries.	Scan for misaligned jump targets [9] and patch the code for analysis (e.g., treat the landing as new code start). Resolve trampolines by inlining or labeling intermediate jumps once identified.
Indirect Memory Addressing	Accesses variables or function pointers via registers (R0/R1/DPTR) instead of direct addresses, concealing which memory locations are used.	Employ symbolic execution or dynamic tracing to determine what addresses pointers resolve to at runtime [12] . Rename memory locations in disassembler once their purpose is inferred. Use read/write breakpoints in an emulator to catch indirect access targets.
Data Masking & Constant Encoding	Hides constant values (keys, IDs, etc.) by storing them encoded (XOR-masked, split, or computed at run-time). Prevents easy identification of important constants.	Identify decoding routines by spotting patterns (e.g., XOR with a constant mask, lookup tables) [10] . Use emulation or symbolic execution to run those routines and recover original constants. Search for known masked data patterns (like encrypted strings) and attempt automated decryption.
Instruction Substitution (equivalent operations)	Replaces common instruction sequences with alternate forms (e.g., arithmetic vs. logical operations) to avoid signatures and confuse decompilers.	Use semantic analysis: e.g., recognize that `XOR A,A` or `SUBB A,A` both clear the accumulator. Pattern-match known alternative instruction idioms. High-level decompilers can sometimes optimize these back to a single operation, aiding recognition.
Mixed C and Hand-coded Assembly	Inserts manual assembly that may break usual conventions (non-standard calling, direct stack manipulation, custom control transfers). Defeats naive decompiler assumptions.	Analyze hand-coded sections in assembly view. Leverage tools with scriptable architecture support (IDA, Ghidra) to mark known library functions [22] and focus on unusual code. Sometimes split the analysis: treat compiled sections with decompiler, and isolate obfuscation stubs for manual dissection.
Unique or Encrypted Firmware Copies	(Advanced) Each device has a uniquely shuffled or encrypted firmware that only the device can decrypt (e.g. via PUF or key in hardware) [17] [18] . Cloning or static analysis yields unusable code.	If hardware-based (PUF) – very hard to reverse without the device. Analysts might attempt to extract the decryption routine from the firmware and emulate it with the key. If instruction order is shuffled, identify the reordering mechanism (e.g., a table of swapped addresses) and reconstruct the correct order by reversing the algorithm (often requires insider knowledge or significant analysis of the bootloader).

Reverse-Engineering Techniques for Obfuscated Firmware

Reverse engineering obfuscated 8051-family firmware is challenging but can be approached with a combination of static analysis, programmatic analysis (lifting to IR, symbolic execution), and dynamic testing. Analysts often combine multiple techniques and tools to undo or see through the obfuscation:

- **Static Disassembly and Decompiler Analysis:** Tools like IDA Pro and Ghidra have support for 8051 disassembly (with community plugins extending decompilation support [23] [24]). The first step is usually to load the firmware binary in such a tool, identify the correct loading address and processor settings, and produce an initial disassembly. However, as noted, heavy obfuscation or even an unusual architecture can thwart disassemblers (Nozomi researchers found IDA couldn't find cross-references in a weird 16-bit firmware until they identified the correct memory model and fixed scrambled jump tables [25] [26]). With 8051 code, the analyst may need to specify the code segment addresses (since 8051 code could be relocated in external memory) and possibly define interrupt vectors so that the disassembler knows entry points. One available tool, **at51** by 8051Enthusiast, can help by automatically guessing the firmware's load offset and analyzing jump patterns [27] [28]. Static analysis can recover a lot: strings, vector tables, and obvious routines. But when obfuscation is present (e.g., opaque jumps, encoded constants), the static view may look like nonsense or have large chunks of code with no clear meaning. At that point, reverse engineers turn to more advanced analyses.

- **Lifting to Intermediate Representations (IR):** Converting the binary into a higher-level IR facilitates the use of modern program analysis techniques. Notably, researchers have created lifters for 8051 machine code into IRs like **LLVM IR and VEX** (used by Angr) [29] [30]. *FirmUSB*, a 2017 framework for USB firmware analysis, implemented the first known 8051 lifters to these IRs [29]. By lifting, the 8051's quirky Harvard architecture and memory-mapped registers are translated into a single unified model that tools can reason about symbolically. For example, FirmUSB's lifter had to map all 256 opcodes and the multiple memory regions of 8051 into VEX's RTL and LLVM's IR types [2] [31]. Once in IR, analyses like control-flow graph recovery, data flow tracking, and program slicing become easier to perform uniformly. IR also enables **cross-architecture** approaches – one can apply tools originally made for x86/ARM to the lifted 8051 code. In practice, a reverse engineer using IR might, say, lift the obfuscated firmware and then run an automated deobfuscation pass or use an SMT solver on the IR to simplify opaque predicates.

- **Symbolic Execution:** Symbolic execution is a powerful technique for analyzing obfuscated code, especially to **solve for hidden constants or trigger conditions**. In symbolic execution, inputs (or certain memory values) are treated as symbols rather than concrete values, and the execution engine explores the program paths with those symbols. This can automatically navigate complex obfuscated logic by treating it as a set of constraints. For example, a checksum or encryption routine in the firmware that's hard to invert manually can be fed to a symbolic executor; the solver can determine what input produces a desired output (useful for unlocking features or bypassing checks). In the context of 8051, symbolic execution has been used to vet USB firmware for malicious behavior [32] [33]. The FirmUSB project combined 8051 IR lifting with both **Angr** (for VEX) and a custom KLEE-based executor called **FIE** (for LLVM IR) [34] [35]. They had to add support for 8051's peculiarities like multiple memory spaces and interrupts [36] [37]. With these in place, they could symbolically execute firmware to, for instance, find what conditions trigger a re-enumeration of a USB device as a HID keyboard (a BadUSB behavior) [4] [38]. Symbolic execution is especially useful to reverse *data obfuscation*: if a constant is encoded via some arithmetic, one can mark the encoded bytes as symbolic and constrain the final result to equal the known plaintext to solve for the mask. It can also systematically explore flattened control flow by forcing the exploration of each branch of an opaque predicate. The downside is path explosion – obfuscated code may have exponentially many paths. Tools mitigate this with heuristics (e.g., FirmUSB pruned paths and scheduled interrupts in a controlled way [39] [40]).

- **Dynamic Analysis and Emulation:** When static approaches falter, reverse engineers may resort to dynamic analysis – i.e. *running the firmware* and observing its behavior. Since running code from a microcontroller often requires simulating the hardware environment, analysts use emulators or instrumented devices. For 8051, options include MCU simulators (like MCU 8051 IDE or MAME's 8051 core) and even QEMU (though vanilla QEMU doesn't widely support 8051, there are patches or specialized forks). By emulating the SN9B292B firmware, for example, one could feed it crafted USB requests or inputs and monitor how it responds. **Fuzzing** falls under this dynamic approach: the firmware's inputs (e.g., USB packet fields, HID commands, etc.) can be fuzzed to see if any unexpected behavior or code paths emerge. A fuzzer might reveal hidden commands or debug modes in the device firmware if not properly obfuscated. (One research, IOTFUZZER, automatically fuzzed IoT device interfaces to find memory corruptions [41] [42], though typically on Linux-based firmware – similar principles apply to microcontroller firmware via instrumentation). In dynamic analysis, **monitoring memory and register state** is key. Tools can set breakpoints on suspicious memory addresses (like the start of an obfuscated constant table) or on specific opcodes (like a `MOVC A,@A+DPTR`, which is often used to fetch constants from code space [43]). By stepping through at runtime, an analyst can bypass confusion in static view – e.g., when an indirect jump occurs, you let it execute and see where it actually goes, then label that destination accordingly in the static disassembly.

- **Pattern Recognition and Automated Aids:** Despite obfuscation, many firmware images still contain chunks of *standard code* (initialization routines, math functions, protocol handlers) that can be recognized. Tools use pattern matching to find these and thereby reduce the unknown portions. For instance, 8051 compilers often include standard library routines (like `memcpy`, `divide`, or hardware drivers). The **at51** toolkit's `libfind` utility can scan a firmware for byte patterns matching known 8051 library functions from common compilers (Keil C51, SDCC, etc.) [44] [45]. Finding these not only labels those functions (so the reverse engineer doesn't waste time on well-known code), but also helps align the code correctly (since library functions have well-defined instruction sequences). Pattern recognition is also used to detect obfuscation patterns themselves. For example, if an obfuscator always encodes constants with the same algorithm, one can write a signature for that sequence of ops. The FirmUSB study demonstrated a form of pattern recognition by scanning the lifted IR for known USB descriptor access patterns – they looked for instructions like `MOV DPTR, #constant; MOVC A, @A+DPTR` which clearly indicate a data pointer is being loaded with a fixed address and then data is fetched from that code address [46] [43]. By finding such patterns, they automatically identified references to USB descriptor tables in the firmware, despite the absence of symbols. Reverse engineers frequently create their own IDA or Ghidra scripts to highlight patterns (e.g., sequences that push `PSW` and manipulate it in a certain way might indicate an opaque predicate check). Modern machine learning approaches even attempt to classify chunks of binary as likely "obfuscated" vs "compiler-generated" by learning from large datasets (though not yet common for 8051 specifically).

- **Firmware Modification and Patching:** As the ultimate step, reverse engineers will often *modify* the firmware to assist analysis. This can mean patching out the obfuscation. For example, if a firmware uses a complex checksum to prevent tampering, an analyst might NOP out the check or replace it with a trivial one and then observe device behavior without that hurdle. Another common trick is to patch the firmware to produce *observable outputs*. Since 8051 MCUs lack rich debugging interfaces, one can repurpose an I/O pin or UART: insert code that, say, writes interim values to a serial port or blinks an LED when certain branches execute. This is akin to instrumenting the binary for insight. Tools like Ghidra allow binary patching; one could use an 8051 assembler to write a small snippet and inject it at some free space in the firmware, then adjust a jump to execute it. While risky on a real device (one must also fix any checksum or integrity checks), this can yield invaluable understanding of an obfuscated routine by essentially adding "printf" debugging. In less invasive scenarios, patching is used to *simplify* the code: e.g., once you deduce that a particular opaque predicate is always true, you can patch the conditional jump to always take the true branch, thus linearizing the code for further analysis. There are also specialized firmware analysis frameworks that support interactive rewriting – for instance, angr's platform can hook specific addresses and execute custom Python code instead, which can be used to skip over obfuscated segments during a symbolic execution.

Case Application (SN9B292B): The SN9B292B USB controller (an 8032 MCU) likely employs several of the above techniques to protect its firmware. USB controllers often carry proprietary logic for USB protocol handling, power management, or even user-interface functions (for example, in a gaming mouse or keyboard). If one were reversing such firmware, one might encounter a flattened main loop handling USB requests (control-flow obf), values like USB descriptors or HID report lengths stored in encrypted form (data obf), and lots of indirect jumps for state handling (e.g., via a state machine pointer). A reverse engineer would first dump the firmware (perhaps via decapping or using a debug interface if available), then load it in IDA or Ghidra with an 8051 profile. Noticing the obfuscation, they could use the techniques above: identify known initialization code (power-on resets for 8051 typically zero certain registers – easy to spot), find the USB request handler dispatch table (maybe by looking for the USB standard request codes in the data), and then mark regions that are garbled. By applying symbolic execution with a tool like angr (augmented with the 8051 VEX lifter from FirmUSB) [47] [48], they could systematically try to follow the firmware's handling of a specific USB command, solving any obfuscated conditions to see what triggers a hidden functionality. Simultaneously, they could run the firmware in an emulator with a virtual USB host to issue commands and verify behavior. Each discovered piece (say, an encoded constant for a descriptor) would be fed back into the static analysis (renaming that constant in the disassembly, etc.), gradually lifting the veil of obfuscation.

Conclusion

Protecting 8051-family firmware with software obfuscation is a double-edged sword: it can significantly raise the bar for reverse engineers, but it also complicates legitimate debugging and maintenance. Techniques like control-flow flattening, opaque predicates, and instruction aliasing have been adopted in both malware and commercial firmware to guard intellectual property [3]. On the other side, the reverse-engineering community has developed equally sophisticated methods – from IR lifting and symbolic execution [29] [30] to pattern-matching tools for known code [22] – to peel back these layers. In summary, firmware obfuscation on 8032/8051 microcontrollers involves a cat-and-mouse game of hiding code meaning and uncovering it. With devices like the SN9B292B, which may control critical USB functions, understanding these obfuscation and de-obfuscation techniques is key for security analysts to find vulnerabilities and for developers to gauge the strength of their code protections. As both industry practices and academic research evolve, we expect to see even more advanced obfuscation (e.g. per-device unique firmware, integrated cryptographic protections) – and correspondingly advanced reverse-engineering frameworks – in the realm of embedded systems security.

Sources:

- Nozomi Networks Labs – *Reverse Engineering Obfuscated Firmware for Vulnerability Analysis* (Mar 2022) [3]

- 8051Enthusiast – *at51 Toolkit Documentation* [9] [22]
- FirmUSB research (CCS 2017) – M. M. Fernandes et al., *Vetting USB Device Firmware using Domain Informed Symbolic Execution* [29] [12]
- Embedded Flakes – *8051 Code Protection Techniques* [49]
- Auburn Univ. research – B. Cyr et al., *Firmware Obfuscation Method for Protecting Systems from Cloning* (IoT Journal 2019) [17] [18]
- dProtect Obfuscator – *Constants Obfuscation documentation* [10] [11]
- Others as cited inline above.

---

[1] [2] [4] [5] [12] [13] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [43] [46] [47] [48] FirmUSB: Vetting USB Device Firmware using Domain Informed Symbolic Execution
https://www.cise.ufl.edu/~butler/pubs/ccs17-firmusb.pdf

[3] [25] [26] How to Reverse Engineer Obfuscated Firmware for Vulnerability Analysis
https://www.nozominetworks.com/blog/reverse-engineering-obfuscated-firmware-for-vulnerability-analysis

[6] Breaking Control Flow Flattening: A Deep Technical Analysis
https://zerotistic.blog/posts/cff-remover/

[7] intro to opaque predicates notes - GitHub Gist
https://gist.github.com/alexander-hanel/3813712fdd1023eee3787da36edf959c

[8] [PDF] Opaque Predicate: Attack and Defense in Obfuscated Binary Code
https://faculty.ist.psu.edu/wu/papers/dongpeng-xu-dissertation.pdf

[9] [22] [27] [28] [44] [45] GitHub - 8051Enthusiast/at51: Applications for reverse engineering architecture 8051 firmware
https://github.com/8051Enthusiast/at51

[10] [11] Constants Obfuscation | dProtect Documentation
https://obfuscator.re/dprotect/passes/constants/

[14] [49] 10 8051 Code Protection Techniques to Safeguard Your Genius - Embedded Flakes
https://embeddedflakes.com/8051-code-protection-techniques/

[15] [16] [23] [24] Science And Engineering - Vakumat P80C552 - Ghidra, BinaryNinja, IDA or Cutter?
https://www.janbeck.com/cybersecurity-and-reverse-engineering-fun/vakumat-p80c552-ghidra-binaryninja-ida-or-cutter

[17] [18] [19] [20] [21] Low-Cost and Secure Firmware Obfuscation Method for Protecting Electronic Systems From Cloning
https://www.eng.auburn.edu/~uguin/pdfs/IoTJ-2019.pdf

[41] [PDF] Discovering Memory Corruptions in IoT Through App-based Fuzzing
https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-1_Chen_paper.pdf

[42] Guide on IoT Firmware Reverse Engineering [Complete Process]
https://www.apriorit.com/dev-blog/reverse-reverse-engineer-iot-firmware