IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 3, pp. 360–389. DOI:10.46586/tches.v2021.i3.360-389

Cutting Through the Complexity of Reverse Engineering Embedded Devices

Sam L. Thomas, Jan Van den Herrewegen, Georgios Vasilakis, Zitai Chen, Mihai Ordean and Flavio D. Garcia

University of Birmingham, Birmingham, United Kingdom
{s.l.thomas,jxv572,gxv724,z.chen,m.ordean,f.garcia}@cs.bham.ac.uk

Abstract. Performing security analysis of embedded devices is a challenging task. They present many difficulties not usually found when analyzing commodity systems: undocumented peripherals, esoteric instruction sets, and limited tool support. Thus, a significant amount of reverse engineering is almost always required to analyze such devices. In this paper, we present Incision, an architecture and operating-system agnostic reverse engineering framework. Incision tackles the problem of reducing the upfront effort to analyze complex end-user devices. It combines static and dynamic analyses in a feedback loop, enabling information from each to be used in tandem to improve our overall understanding of the firmware analyzed. We use Incision to analyze a variety of devices and firmware. Our evaluation spans firmware based on three RTOSes, an automotive ECU, and a 4G/LTE baseband. We demonstrate that Incision does not introduce significant complexity to the standard reverse engineering process and requires little manual effort to use. Moreover, its analyses produce correct results with high confidence and are robust across different OSes and ISAs.

Keywords: Reverse engineering · Embedded device firmware · Hardware-based execution tracing

1 Introduction

When performing security analysis of end-user embedded devices, reverse engineering is often required to gain a deep understanding of a device and its firmware. In general, this is a challenging and time-consuming task, even for experts. The core difficulties stem from how devices are composed and how tooling to simplify analysis can only go so far in terms of what can be automated. Developing tooling that generalizes over multiple devices is complicated and prone to scalability issues. Most moderately complex consumer devices are composed of numerous peripherals and CPU cores that interact with each other and their environment, often under hard real-time constraints, further complicating the process.

Recent work has sought to address the challenges involved in performing analysis by emulating firmware and peripherals [CGS+20, FML20, GMS+19, MFL+21]. These approaches enable device firmware to be *rehosted*—either fully or partially—within a commodity emulator and then analyzed without interacting with the physical hardware. However, to rehost real firmware, it is often necessary to perform a non-trivial amount of reverse engineering of the device, its peripherals, and its firmware. Such techniques are, therefore, not always practical when time is a limiting factor in performing analysis.

Other techniques address the challenges by incorporating the device *in the loop* during analysis [CCF18, CF20, MFB18, ZBFB14]. Though without careful harnessing, for real-time constrained and complex bare-metal firmware, the timing overheads induced by these

Licensed under Creative Commons License CC-BY 4.0.
Received: 2020-10-15 Accepted: 2020-12-15 Published: 2021-07-09

approaches can lead to unpredictable behavior in the target device due to violations of hard real-time constraints [KKM15] and interference with timers. To identify the source of these violations and develop suitable harnesses, we must again resort to reverse engineering the device and its firmware.

Table 1 lists a broad selection of recently published articles proposing techniques for analyzing embedded firmware. Irrespective of the analysis goal or approach used, when applied to real-world targets, all require a non-negligible amount of manual reverse engineering to perform. Without this upfront reverse-engineering, we would not know how to interact with the firmware or which parts of it to analyze. Thus, developing methods to reduce this reverse engineering effort is essential to use such automated analysis techniques to analyze end-user devices.

Table 1: Embedded firmware analysis approaches. Amount of prior reverse engineering (R-E) measured by: ✚ for identification of interrupts and handlers, stall states, and context switches, ✚✚ for peripheral interaction, cross-core communication, and ✚✚✚ for specific functionality (for stubbing/harnessing). Requirements refer to target support needed: (C)ompiler, (D)ebugger, (DI)sassembler, (E)mulator, (S)ource Code, (T)racer.

	R-E	Requirements						Objective	Enabling mechanism
		C	D	DI	E	S	T		
Avatar [ZBFB14]	✚✚	○	●	●	●	○	○	Dynamic analysis	Hybrid execution
BaseSAFE [MSP20]	✚✚✚	○	●	●	●	○	○	Fuzzing	Partial rehosting
DICE [MFL+21]	✚	●	●	●	●	●†	○	Fuzzing	Peripheral modeling
Frankenstein [RCGH20]	✚✚✚	○	●	●	●	○	○	Fuzzing	Partial rehosting
HALucinator [CGS+20]	✚✚✚	○	●	○	●	○	○	Fuzzing	HAL emulation
HardSnap [CF20]	✚✚	●	●	●	●	◐	○	Symbolic execution	Partial rehosting
Inception [CCF18]	✚✚	●	●	●	●	◐	○	Symbolic execution	Hybrid execution
Laelaps [CGML20]	✚	○	●	●	●	○	●	Dynamic analysis	Peripheral modeling
P^{2}IM [FML20]	✚	●	●	●	●	●†	○	Fuzzing	Peripheral modeling
Pretender [GMS+19]	✚	○	●	●	●	○	●	Rehosting	Peripheral modeling
Surrogates [KKM15]	✚✚	○	●	●	●	○	○	Dynamic analysis	Hybrid execution
Incision ‡ (this paper)		○	○	●	○	○	●	Reverse engineering	Trace comprehension

† DICE and P^{2}IM require firmware to be compiled with a call to `startForkserver` for fuzzing [FML].
‡ Incision is used in the pre-analysis step to perform initial reverse engineering.

1.1 Our Contribution

In this paper, we address the challenge of reverse engineering complex embedded device firmware to facilitate more straightforward manual analysis and application of automated analyses. We present Incision, a device in the loop reverse engineering framework, based on lightweight execution tracing and static firmware analysis. Our framework is both Operating System (OS) and architecture agnostic. It complements the standard reverse engineering workflow and approaches such as those listed in Table 1, by aiding functionality identification, control-flow recovery, and correct disassembly of binary blob firmware.

Our approach operates in feedback loops that progressively improve the reverse-engineered representation of the firmware analyzed. A single iteration consists of three high-level steps: region inference, trace capture, and trace processing. Each iteration identifies firmware regions containing functionality related to an input reverse engineering policy. We use a machine encoding of the policy as input to a clustering algorithm to perform region inference. We use the outputted regions to decide which portion of firmware to focus on and what to trace. Due to interrupt processing and task switches, system-wide traces often end up being fragmented. Therefore, to recover control-flow information, we identify task switching logic and interrupt handlers and partition the traces based on

this information. Then taking the resulting sub-traces, we group those that follow the same execution flow and combine them back into constituent traces. Finally, we refine our firmware representation with recovered control-flow information from those traces. We add new control-flow edges during refinement and use them to locate new functions, static data references and rectify incorrect disassembly. The process leverages statically inferred and dynamically witnessed control-flow information in tandem. At the end of the analysis, Incision outputs the bounds of functionality corresponding to our reverse engineering policies and a firmware representation suitable for further analysis.
In summary, our contributions are:

1. An analysis of the specific challenges faced when reverse engineering complex embedded devices and their firmware.
2. Four techniques for automating aspects of the reverse engineering process: ❶ Firmware region inference. ❷ Identification of OS primitives for task and context switching from execution traces. ❸ Task-aware control-flow recovery from system-wide execution traces containing multiple interleaved tasks and interrupts. ❹ A feedback mechanism that combines the results of the previous techniques to improve our firmware representation and rectify disassembly errors.
3. An evaluation of Incision against two complex real device firmware (an LTE baseband and an automotive Body Control Module (BCM)) and a set of firmware simulating a range of device configurations. In full, our evaluation spans three different Real-Time Operating Systems (RTOSes), VxWorks, FreeRTOS, and Zephyr, a bare-metal OS, and two different instruction sets (ARM/Thumb2 and Renesas V850ES).

To aid future research, we release our implementation and data sets as open-source at https://github.com/UoBAutoSec/INCISION.git.

2 Background

In this section, we establish the scope of devices and firmware analyzed. We outline their composition, the standard approaches to analyze them, and the challenges faced.

The target of our approach is *hard to analyze* embedded device firmware. We define this as monolithic interrupt-driven software operating with or without an OS. For firmware with an OS, we restrict the definition to firmware that operates under real-time constraints. We refer to firmware without an OS as *bare metal*. We call the combination of a device and its firmware a System Under Test (SUT).

The execution of interrupt-driven software is directed by handling and processing interrupts. Interrupts are triggered by (on- and off-chip) peripherals, the OS, and system tasks. A task represents the execution of a part of software or firmware responsible for performing a specific function, such as input processing. Interrupt-driven software is composed of one or many tasks, which together constitute its functionality. An RTOS is a specific type of interrupt-driven software that operates under soft or hard real-time constraints. Time-sensitive tasks orchestrated by the OS must be completed within strict windows to adhere to these constraints. An RTOS manages the execution of many concurrent tasks. Each task can be interrupted by one running at a higher priority, and interrupts may be nested.

2.1 Reverse Engineering Embedded Firmware

Reverse engineering is the process of taking a SUT and determining properties about its inner workings. The principal objective of reverse engineering is to understand the SUT to

explain how it works and extract specific implementation details. Therefore, the process necessarily involves a high degree of human intervention—both during the analysis stage and afterward.

We perform reverse engineering using two complementary classes of analyses: static and dynamic. Static analyses form conclusions about a SUT by analyzing it at rest. In contrast, dynamic analyses allow us to reason about observations of its runtime behavior. Interactive disassemblers such as IDA Pro [Hex] and Ghidra [Nat] are at the core of most reverse engineering approaches. They enable an analyst to interact and manipulate a static representation of firmware to gain a deep understanding of its functionality. Similarly, debuggers and execution tracers fulfill that role for dynamic approaches by enabling us to reason about the SUT's execution at different points.

Interactive disassemblers represent programs using a database. This database typically contains each executable segment's disassembly, a control-flow graph, identified functions, and cross-references between code and data. For commodity software, interactive disassemblers can automatically populate the database. However, for most embedded firmware, database creation involves manual effort. RTOS-based and bare-metal firmware do not have standard container formats and vary widely in their composition. The only commonality is that their application logic, library code, static data, and OS (if present) all exist within the same binary blob. Thus, to populate the database, we need to determine the firmware's memory mapping (i.e., where its regions are loaded) and Instruction Set Architecture (ISA). As interrupt-driven firmware does not have a single entry-point, we must also identify interrupt vector tables, addresses of callbacks for tasks, and other initialization routines.

Disassembly describes the process of translating raw bytes into architecture-specific instructions. A disassembler requires viable instruction starting offsets within firmware to perform correct disassembly. For embedded ISAs, locating these offsets is not a trivial task. Many instruction sets are variable length encoded (VLE) with short instruction widths, so plausible disassemblies exist at many offsets, of which only one is correct. For some architectures, such as ARM/Thumb2, the correct instruction set to use for disassembly is context dependent, further complicating the process. Identified function starts are a common means to locate starting points. However, as non-standard calling conventions are commonplace in embedded firmware, we cannot always rely on known instruction patterns to determine starting points and often must identify the offsets manually.

To analyze the control-flow of firmware, we recover it from the disassembly. Control-flow recovery algorithms reconstruct a Control Flow Graph (CFG) containing the transitions between blocks of disassembled instructions. In this case, a block is a sequence of instructions terminated by a control-flow altering instruction, e.g., a branch or a call. Since interrupt-driven software contains a high degree of indirect control-flow, it is difficult to statically estimate the destinations of many of these branches and calls (a well documented problem [CKB17, DBXP20]). Therefore, static control-flow recovery for embedded firmware often results in highly incomplete CFGs [Fri].

When reverse engineering large and complex software, such as firmware, we must identify where to focus our analysis effort. Symbols (function names) are beneficial for this purpose. However, they are often not present in embedded firmware. Static data (debug strings) are also useful, as we can analyze code that references strings of interest under the assumption that code will reference strings related to its functionality. When all of the data exists in the same section, this is a trivial process. However, in firmware, it is often embedded inside code segments to avoid expensive references to locations at offsets exceeding what can be addressed by a single load instruction. Therefore, when disassembly and control-flow recovery results in an incomplete program database, the amount of data references to guide the reverse engineering process will be similarly limited.

To analyze a device dynamically, we cannot debug it *live* without interfering with how it communicates with its peripherals. To obtain useful analysis results, we must faithfully

simulate a realistic external environment, e.g., by providing its peripherals with input. Without this, the firmware will stall until it receives an expected input. Identifying stall points and handling them is necessary to enable most kinds of dynamic analysis.

When available, execution tracing facilities allow us to work around the problems associated with debugging. Execution tracing involves tracking part of the state of a SUT over an execution. The granularity of this tracking is dependent on the tracing technology available. Many devices offer hardware-based support for execution tracing [ARMa]. While lightweight, this type of tracing has some limitations. Due to hardware constraints, traces are often limited in size and can only capture certain types of events, such as the program counter's value at branches. Under certain conditions, e.g., when firmware executes a tight loop or an interrupt handler, events may get dropped, resulting in fragmented traces. Some tracing technologies support starting tracing on a particular condition. However, we must specify these conditions before beginning tracing. As this kind of tracing is system-wide, it captures events across multiple tasks. Traces, therefore, represent a temporal ordering of events rather than real control-flow.

3 Overview

In this section, we detail the challenges tackled by our approach, Incision. We then list the assumptions made in its design and provide an overview of how it operates. As detailed in the previous section, embedded device firmware introduces significant complications to the reverse engineering process. The challenges below represent the difficulties present when reverse engineering embedded device firmware compared to commodity software:

1. State-of-the-art tools provide limited support for embedded architectures resulting in disassembly errors that propagate through the analysis process.
2. Embedded device firmware is monolithic and often contains no symbol information. When debug information is present, it is usually not located within a single section but mixed with code and data, making it difficult to leverage for firmware comprehension.
3. Many end-user devices offer limited hardware tracing facilities. The available mechanisms are often error-prone and limited in terms of the amount and type of events they can capture.
4. Captured traces of interrupt-driven firmware do not reflect real control-flow and instead represent a temporal ordering of events.
5. Combining static and dynamic analysis methods is difficult due to the strong coupling of hardware and software. Analysis requires a human in the loop to orchestrate trace capture, provide input to the SUT, and reset it in case of error.

We build Incision around Ghidra [Nat], an open-source interactive disassembler. We integrate it into the standard reverse engineering workflow, where static and dynamic analyses are used in conjunction to form conclusions about the SUT. We make the following assumptions about the type of devices and firmware analyzed:

1. We know the ISA of the firmware, its memory mapping, and can identify at least one entry-point.
2. We can trace the execution of the device (e.g., using hardware-based trace facilities). The granularity of traces produced is at least at the basic block-level.
3. Captured traces may contain errors, i.e., dropped events, but no active measures have been taken to restrict tracing. If present, these errors are reported.

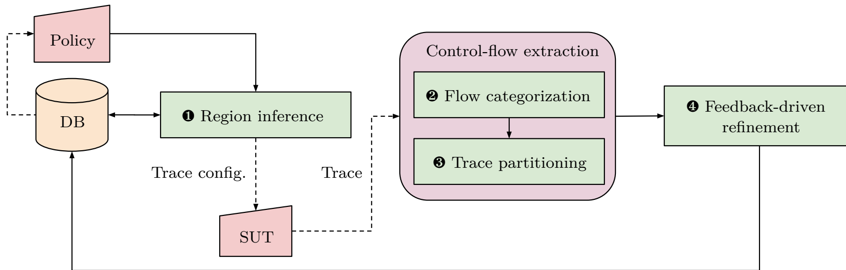

Figure 1: Overview of Incision. We indicate data-flow by block arrows for automatic propagation and dashed arrows for manual propagation. Incision operates in four step feedback loops. Region inference (step ❶) takes a policy and derives bounds to form a trace configuration. We use the bounds to take a trace of the SUT. From the trace, we perform control-flow extraction (steps ❷ and ❸) to identify task-switching logic and obtain task-level CFGs. Finally, we integrate the extracted CFGs into our DB and apply a static analysis pass to propagate the newly added information (step ❹).

Figure 1 provides an overview of our approach. We first import the firmware into Ghidra to produce a program database. We perform region inference (step ❶) based on an initial reverse engineering policy (an encoding of our initial analysis goals) using the database as input. Region inference identifies firmware regions corresponding to the policy. We use these regions' bounds to configure the SUT's tracing mechanism and perform trace capture. From this, we obtain a system-wide trace containing the interwoven control-flow of multiple tasks. To extract each task's control-flow, we analyze the trace in two stages. In the first stage (step ❷), we determine which parts of the trace correspond to task switching logic and interrupt handling. In the second stage (step ❸), we split the trace based on the first stage, by extracting sub-traces separated by task switching or interrupt handling logic. We then recover each task's control-flow by combining sub-traces into a dynamic CFG per task. Finally, we update the program database using the extracted control-flow (step ❹) by performing disassembly of previously unexplored regions and adding control-flow edges. We follow this with a static control-flow recovery pass over the updated program database. This pass allows us to recover additional control-flow edges and data references not identified in traces or the initial database and correct disassembly errors.

We apply steps ❶-❹ iteratively. On each iteration, we refine our database, while at the same time, we *zoom* into the firmware regions corresponding to our reverse engineering goals. Each iteration necessarily requires some human intervention for processes that cannot be automated: to import the firmware into Ghidra, to perform trace capture (as the mechanism for doing so is generally hardware-based), to provide input to our region inference algorithm (which requires encoding of our reverse engineering goals), and to resolve database conflicts we cannot address automatically. After several iterations, our framework's output (firmware database and traces) provides a basis for performing further analyses, such as those listed in Table 1.

4 Methodology

In this section, we describe the technical details of our approach. We discuss each component and the corresponding challenges it addresses.

4.1 Inputs

Incision takes three kinds of input, a program database, execution traces, and reverse engineering policies. A program database corresponds to a Ghidra database. This database

is created by loading the firmware into Ghidra at the correct image-base and triggering a basic *auto-analysis* pass. From this, we obtain an initial set of function starts, a call graph, and an intra-procedural control-flow graph for each identified function.

We base our execution trace format on the ARM ETB format [ARMc]. It captures block-level control-flow event packets and encodes two high-level trace errors. The first kind of error is for trace capture toggles. We use this to flag that event packets are missing because of a trace parameter. The other type of error is for trace mechanism errors. We use it to flag when event packets are missing due to overflows. A common cause of this type of error are events that occur too frequently to be processed by the trace hardware.

We encode reverse engineering objectives as reverse engineering policies. We specify policies as a vector of one or more: keywords, symbol names, raw addresses, and instruction patterns. Keywords match static data, symbol names match program database labels, raw addresses match referents of database cross-references, and instruction patterns match disassembled instructions that have specific syntactic properties. We write symbol and keyword matchers as regular expressions. An instruction pattern may be suffixed with `+` to indicate that it should only yield a successful match when it repeatedly occurs in a trace (e.g., as part of a loop). The policy ⟨`(mov ?, 0xcafe)+`⟩, for example, matches instruction locations where the constant `0xcafe` is used as a source operand in a `mov` operation repeatedly over a trace. At the same time, ⟨“Rrc”, “RrcSmc”, `memcpy`⟩ encodes the goal of locating `memcpy` usage within a region with references to static data containing the strings “Rrc” and “RrcSmc”. Policies are used as input to our region inference and feedback-driven refinement algorithms. They evolve across each iteration of Incision to become finer-grained, reflecting our increased understanding of the SUT over time.

4.2 Region Inference

To determine what to trace and which part of the firmware to focus on, we perform region inference. This process locates firmware region boundaries corresponding to functionality specified by the input reverse engineering policy. We perform inference in two steps: region identification and region grouping (shown in Algorithm 1, Appendix A). Region identification locates disparate regions (functions) corresponding directly to the input policy, while region grouping locates the boundaries of encompassing regions and includes transitively related functionality. Grouping is required as many embedded tracing mechanisms are limited in the number of tracing regions (inclusions or exclusions) that can be specified. By locating larger encompassing regions, we, therefore, ensure traces are complete and specific and minimize overheads due to entering and leaving trace regions, which can cause trace errors.

4.2.1 Region Identification

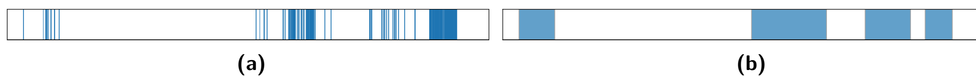

Figure 2: (a): Region identification applied to LTE firmware using a policy of ⟨“NAS”⟩. **(b):** Region grouping applied to the result of (a) for $R = 4$.

We perform region identification by identifying functions containing elements or references to elements matching the input policy. We then group the functions by their locality and rank them based on the number of times each group matches each input policy element. We consider references to be in the same group if they occur within δ bytes of each other; we set δ as twice the database's median function size. We rank policy elements associated

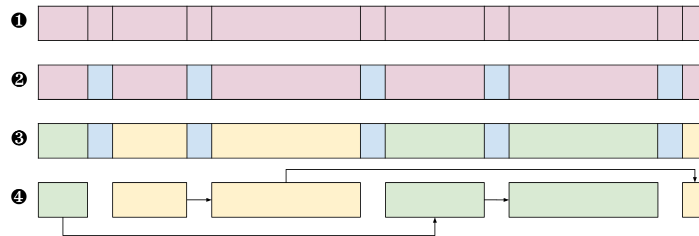

Figure 3: Visualization of control-flow extraction for two tasks. We first identify contiguous regions of viable control-flow in the input trace (step ❶), then identify task switching and interrupt handling logic (shown in blue in step ❷), identify the control-flow for each task (task 1 shown in green and task 2 in yellow in step ❸), and finally, perform task-level CFG extraction (step ❹).

with each region by the number of times they are referenced or occur, forming a ranking of all referencing regions of an element based on their relative frequency. This ranking metric is based on the observation that the functionality of a region which heavily references a particular element is likely to be closely related to that specific element. Since we match static data and symbols based on both full and partial matches, we treat strongly related referenced data as a single entity. By doing so, we limit the amount of static data, such as debug strings, considered in isolation due to being embedded near the code that uses it and only references it once.

4.2.2 Region Grouping

Region identification locates sections of firmware containing functionality specific to a given policy. However, we may identify more regions than our trace mechanism can handle if the functionality is split over non-contiguous regions separated by more than δ bytes. We show such a scenario in Figure 2a, which depicts region identification applied using the policy ⟨“NAS”⟩ to an LTE baseband firmware. To overcome this, we combine the identified regions by grouping them into R larger regions.

To perform grouping, we use hierarchical agglomerative clustering [War63] with proximity as the distance metric. This type of clustering works by iteratively merging regions based on their proximity until R remain. We base our grouping method on the observation that functions with related functionality (i.e., that are used together in the same module to provide some higher-level functionality) are generally located close together in program binaries. We stipulate that this is due to how software is developed—the standard practice is to put related code into the same module, and this grouping gets preserved during compilation. Figure 2b shows region grouping applied to the result of region identification with $R = 4$. The clustering groups the disparate regions in Figure 2a into four larger regions containing related functionality.

4.3 Control-Flow Extraction

System-wide traces contain a mix of application-level code (i.e., task code), interrupt handling, and OS-level task switching logic. To extract the control-flow for each of these sub-traces, we use the CFG representation from the program database to *follow* the execution of the trace. We first identify continuous flows of blocks such that there exist *real* control-flow transitions between each contained block. We then categorize blocks we could not determine viable transitions to or from, using the following labels: *breaks* for context-switch like behavior, *unresolved* for feasible transitions we could not yet resolve due to an incomplete database (e.g., indirect flows), and *errors* for blocks that do not

exist within the database. Based on this categorization, we split the trace into flows that correspond to either task code and context switching logic. This enables us to isolate the control-flow of each trace part, and extract each task's control-flow. We visualize the complete process in Figure 3.

4.3.1 Flow Identification

We identify flows (visualized by step ❶ in Figure 3) by performing a single pass over the input trace. We process each block pairwise with its successor. For each pair of blocks, we use Algorithm 2 (Appendix A) to categorize the transition between them as either *viable*, an *error*, or *unresolved*. We use the *error* label for transitions where either block failed to disassemble. We use *viable* when there exists an irrefutable flow between the blocks due to a branch, call, or return. To handle returns, we maintain a call stack by pushing the address following a call onto the stack and popping the top of the stack on each return. We consider a return between two blocks *viable* if the address at the top of the call stack matches the second block's address. Since a trace may contain events for several tasks, we cannot rely on it to help us resolve indirect control-flow. Thus, at this stage, we conservatively handle indirect calls. To do so, we propagate constants referenced in the function enclosing the first block. If this enables us to resolve the call-site target to the second block's address, we label the transition *viable*. We handle additional cases later when performing control-flow graph extraction (Section 4.3.3).

When we encounter a tracer-induced error between two blocks, it indicates that at least one block is missing from the trace. If context switching occurred between the blocks, then the second will not be reachable from the first by regular control-flow. However, if there should be a *viable* transition between the blocks, then there will be some path in the program-wide CFG connecting them in some bound N of intermediate blocks. To capture this, as we process traces, we construct a dynamic CFG containing transitions between blocks across untraced regions. We then use this CFG to approximate *viable* transition targets when errors are present. As this may lead to unsoundness when we label error transitions as *viable*, to minimize false positives, N must be small; for all firmware analyzed herein, we set $N = 3$. When no viable flow can be identified based on the current database, we terminate the current flow and classify the transition as *unresolved*.

4.3.2 Flow Classification

To determine which flows correspond to task code or context-switching, we identify which *unresolved* transitions are *from* task-switching logic and *to* interrupt handlers that should be re-classifed as *breaks*. This process enables us to find out where task code starts and ends. To perform re-labelling, we analyze how the prefixes and suffixes of flows repeat across the entire trace. We base this on the observation that task switching and interrupt handling code exhibits execution patterns distinct from regular code. We use a suffix array to represent the input trace and an Longest Common Prefix (LCP) array to locate repeated prefixes and suffixes, constructed using Algorithm 3 (in Appendix A). An LCP array enables us to determine the length of the common prefix between two sub-sequences (or flows).

Neither suffix arrays nor LCP arrays are directly amenable for representing trace data. Thus, we perform some adaptations. Suffix arrays operate on sequences taken from a fixed alphabet Σ. For traces, we use an alphabet derived from block start addresses. We consider two blocks with the same address distinct if a trace error precedes or follows them. Suffix arrays also require us to signify the end of an input sequence, a so-called sentinel character $\Sigma^{\perp}$. For traces, we select this character as $\max(\Sigma) + 1$. By design, an LCP array enables us to find *all* repeated sub-sequences of an input sequence, including those that overlap. However, for execution traces, only those that do not overlap are meaningful. We

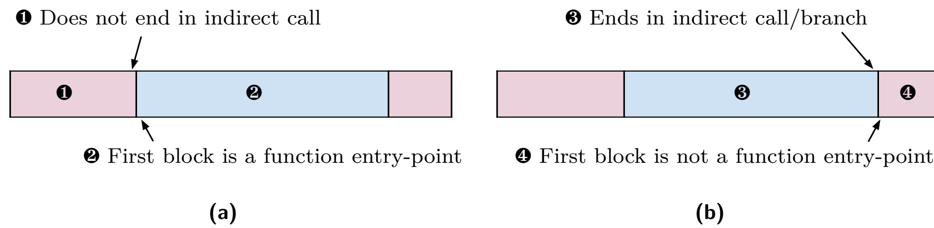

Figure 4: (a) When there is no viable control-flow between two flows (e.g., where ❶ does not end on an indirect call and ❷'s first block is a function start), we classify the second flow as an ISR (❷ in blue). **(b)** When there is viable control-flow between two flows, but it is not regular (e.g., where ❸'s last block ends on an indirect call and ❹'s first block is not a function start), we classify the first as task-switching logic (❸ in blue).

thus compute the length of the common prefix between two sequences as the minimum of the reported LCP length and the difference between their indices in the suffix array.

Identifying Transitions to Interrupt-Handling Logic We identify the starting points of context-switching code by identifying transitions to ISRs. Our intuition for this is based on the fact that task switches as well as other kinds of context-switching are typically preceded by an Interrupt Request (IRQ) (e.g., a timer interrupt preempting a certain task). IRQs are then handled by an ISR or interrupt handler, which determines how the interrupt will be processed.

To locate ISRs in a generic way, we first identify pairs of neighboring flows, where ❶ the last block of the first flow does not end in an indirect call or branch, and ❷ the first block of the second flow is the entry-block of a function, as shown in Figure 4a. To distinguish transitions to ISRs from other *unresolved* transitions matching this criteria, we filter them by only considering those that have execution patterns that appear frequently in the trace (which we identify using the LCP array). We detail the complete procedure in Algorithm 4 in Appendix A.

Identifying Transitions from Task-Switching Logic Similar to interrupt-handling logic, we locate task-switching code by identifying pairs of neighboring flows with specific properties, visualized in Figure 4b. Before task-switching happens, a task context will be set-up or restored from a global structure; then, control will transition to the newly created or resumed task using an indirect branch or call. We can distinguish these calls and branches from regular indirect calls and branches by their targets. For task-switching logic, the set of possible targets is unbounded, and they are unrestricted in their locality. Further, they can be function starts, block starts, and the middle of basic blocks. The number of firmware locations for performing task switching is small (usually one) and frequently appears within traces.

Thus, to identify task-switching logic, we construct an ordering over the last block of each identified flow. We consider the highest-scoring block to be part of the flow performing task-switching. We score the blocks based on the following criteria: ❶ (greater) if the block occurs at the end of more than one flow and is followed by blocks that are function starts and blocks within functions, ❷ (greater) if there is no trace error between the block and the next, ❸ the total number of times the block occurs at the end of a flow, ❹ the number of unique blocks that follow the block, and ❺ the length of the longest common suffix of the flows terminated by the block (computed using the LCP array). Criteria ❺ reflects that set-up code for task switching is mostly branch free.

4.3.3 Task-Aware Control-Flow Extraction

To extract task-level control-flow, we first merge flows of the same kind (task-switching logic or task code). To do this, we use the labels applied to flows and transitions in the previous phases. We then merge non-adjacent flows to build flows for each task. Finally, we output the task-level control-flow graphs derived from the merged flows.

We always elect to merge adjacent flows that are both labeled as task-switching logic, which occurs when the task-switch code is itself interrupted (nested interrupts). We merge adjacent task flows separated by *unresolved* transitions if the first flow ends with a computed call (e.g., `call [R0, #offset]`). We also merge flows if the second flow starts at the entry of a function or if the first flow ends on a return and the second starts at a block whose predecessor ends on a computed call target.

To merge non-adjacent flows and hence detect task-resumption, we maintain a stack of flows while merging. At each point following an identified task switch, we attempt to merge the next flow with one of the unmatched flows in our stack. We consider two flows to match when the second starts (or resumes) in the same block as the first ends on, or when the second starts in a block reachable within a single transition from the block that the first ends on, modeling resumption into a successor block. When a trace mechanism drops events, identifying task resumption is generally not possible because the ends and starts of flows may miss intermediate blocks. Further, due to code-sharing between tasks, and as traces only contain basic block starts, it is not always possible to discern which task resumed if the task-switch occurred when executing shared code (e.g., library code). We detail the complete procedure in Algorithm 5 in Appendix A.

4.4 Feedback-Driven Refinement

At the end of each iteration performed by Incision, we add control-flow edges to the program database based on extracted task-level CFGs, correct disassembly errors, and propagate that information across the entire database. We detail the full procedure in Algorithm 6 in Appendix A.

We apply Ghidra's auto-analysis to perform propagation, which identifies new function starts and code/data cross-references. Propagation acts as a feedback-mechanism. It enables us to incorporate reverse engineering policy-specific information extracted from the input trace (such as control-flow edges) and use it to drive further static analysis-based refinement, which will discover additional functions and cross-references.

To handle disassembly errors identified while processing the trace, we re-disassemble blocks using alternative disassembler options. For ARM, for example, we switch to/from Thumb mode. To ensure that incorrect disassembly does not propagate across the database, we re-disassemble blocks transitioning into "error blocks" and force Ghidra to rebuild their enclosing functions to correct the function-level control-flow. This automated process can fail when there is a conflict in the database. For example, when an incoming edge forces disassembly to be performed incorrectly with a specific set of options, or when a function is incorrectly labeled as non-returning causes disassembly to stop and miss code following calls to that function. To remedy this, we require human intervention to correct the error. In practice, the frequency this is required is low, and the intervention is usually minor (Section 5.4).

Following propagation, to locate repeated instruction patterns from our input policy, we label each function with any instruction patterns it matches. To find them, we use the LCP array from the previous phase to locate repeated sequences containing each matched instruction's enclosing block. Upon completion of an iteration, we terminate Incision when our reverse engineering objectives have been satisfied (e.g., we identified the firmware region responsible for a given behavior). Otherwise, we update our reverse engineering policy and perform another iteration of Incision's feedback loop.

5 Evaluation

In this section, we evaluate the effectiveness of INCISION. We first assess each of its components separately: region inference (RI), task-based control-flow extraction (CE), and feedback-driven refinement (FD). Then, we apply INCISION in full by performing real reverse engineering tasks. Our evaluation assesses *correctness*, *real-world usability*, and *human effort* using the following criteria:

C.1: Correctness. ❶ For CE, the extent to which INCISION extracts correct control-flow of input traces, its ability to discern between task and context-switching logic, and the correctness of the identified flow bounds. ❷ For RI, the extent to which the tracing bounds reported by INCISION correspond to functionality relevant to the input policy. ❸ For FD, the extent to which INCISION improves the firmware database. ❹ For FD, whether INCISION correctly rectifies disassembly errors.

C.2: Real-world usability. ❶ The extent to which input policies reflect real reverse engineering goals. ❷ How effectively INCISION applies to real-world SUTs.

C.3: Human effort. For FD, the proportion of errors that require manual intervention compared to what INCISION can fix automatically, how many changes to the database that equates to and the complexity of those changes.

5.1 Firmware Data Set

We conduct our evaluation using firmware extracted from end-user devices and firmware adapted from open-source projects. Our end-user devices consist of a VxWorks (RTOS) based Huawei LTE baseband targeting ARMv7 and a Renault BCM targeting Renesas V850ES. We use ARM CoreSight to trace the baseband, as detailed in Appendix B. For this particular device, CoreSight's trace capture buffer is limited to 1MB. For the BCM, we perform tracing using Renesas CubeSuite+ [Ren]. Both SUTs are of moderate complexity and interact with numerous peripherals. To establish a ground truth, we evaluate INCISION using two open-source RTOSes, FreeRTOS, and Zephyr. To do so, we select existing example firmware from each project and expand its functionality by adding: encryption and decryption operations, sorting and searching operations using custom predicates (as a means to introduce many indirect calls, which represent an adversarial setting when performing control-flow extraction), and various data encoding and decoding schemes, each running as different OS tasks. In total, we construct ten firmware images, five for each OS. To capture execution traces, we use QEMU.

5.2 Correctness (C.1)

5.2.1 Control-Flow Extraction (C.1.1)

For this criterion, we evaluate using our FreeRTOS- and Zephyr- based firmware. We construct a ground-truth by manually identifying task code and the ISR responsible for triggering task switches from the firmware source code. We then match these locations to regions in the compiled firmware. In the Zephyr-based firmware, the context switching starts with the clock interrupt `z_clock_isr` and for FreeRTOS-based firmware it starts with `xPortSysTickHandler`. In both OSes, the context switch ends with a `PendSV` exception. For each firmware, we vary the number of concurrent tasks from two to ten. We also configure the scheduling mechanism of each OS to vary the frequency of task switches from 1ms to 6ms.

Experiment: We trace each firmware and sample five non-overlapping sub-traces of 50k blocks from each trace with a different task switch frequency. Then, we compare the identified tasks, resumes, and switching logic to our ground truth.

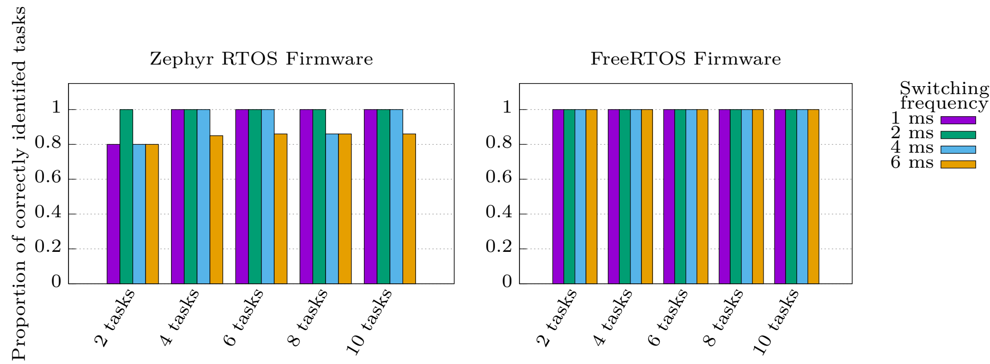

Figure 5: Left: Correctly identified tasks for Zephyr firmware; **Right:** Correctly identified tasks for FreeRTOS firmware. We group the results by task-switching frequency (1ms, 2ms, 4ms, and 6ms). We calculate the resulting correctness for each group as the average correctness over five trace samples of 50k blocks.

Results: Figure 5 shows the proportion of correctly identified tasks averaged over five traces for each firmware. In over 80% of the cases, INCISION correctly identifies the task switching logic. Furthermore, when it correctly identifies the task switching behavior, INCISION can correctly discern between task starts and task resumes. The incorrectly identified task switches stem from a limited number of switches in the trace samples. The application-level indirect control-flow outnumbers the number of task switch targets, thus inducing false positives. This would not be an issue in practice, as traces from real-world devices have a higher prevalence of task-switching due to peripheral interaction.

5.2.2 Region Inference (C.1.2)

For this criterion, we assess the quality of the region bounds output by INCISION using the policies shown in Table 2. We use the Huawei LTE baseband for this experiment, as it is the most complex SUT in our data set. We construct a ground truth by manually identifying regions based on symbols and debug strings in the firmware. We further support this evaluation in Section 5.3.2 by taking traces using the identified bounds for a real reverse engineering task.

Experiment: We measure quality by calculating the overlap between the regions identified in our ground truth and to those output by INCISION. For each policy, we limit the number of regions to find to four to match the number of regions traceable using the SUT's CoreSight configuration.

Results: Table 2 shows that all of the ground truth bounds fall within regions output by INCISION, with at least 79% overlap. As demonstrated later in Section 5.3.2, these bounds enable us to acquire traces containing functionality matching the specified input policies. Further, those traces enable us to perform practical reverse engineering. In this way, INCISION remedies some of the limitations of hardware-based tracing mechanisms, such as trace buffer limits. It can narrow the number of regions to trace while still enable the capture of relevant functionality.

5.2.3 Database Improvement (C.1.3)

To assess this criterion, we compare the increase in identified function starts after each iteration of INCISION to a baseline of Ghidra's auto-analysis. We use our real-world firmware for this experiment. To establish a ground truth of function starts for the baseband, we use leaked firmware symbol information. For the BCM, we establish a ground truth by manually reverse engineering its firmware.

Table 2: Comparison of size and ratio of overlap of manually identified regions compared to regions identified by Incision for policies used in Section 5.3.2.

Policy	Area of ground truth region	Area of overlap with identified region	Ratio of overlap
⟨“USIMM”⟩	95009	91677	96.49%
⟨“WAS”⟩	1412691	1116338	79.02%
⟨“NAS”⟩	946908	904951	95.57%
⟨“RRC”⟩	792107	722861	91.26%
⟨“RRC_SMC”⟩	9086	8524	93.81%

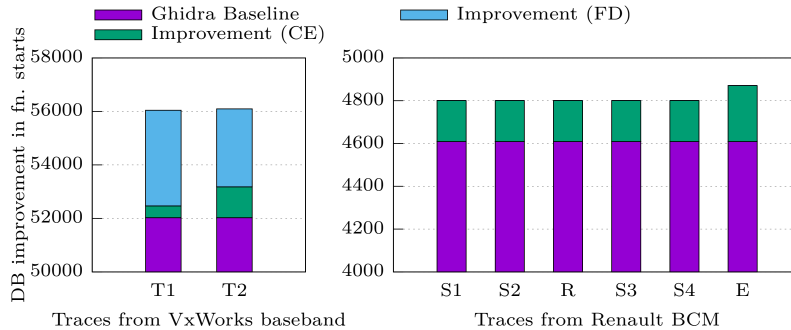

Figure 6: Left: Database improvement for VxWorks baseband firmware; **Right:** Database improvement for Renault BCM firmware. Improvement measured in correct function starts; categorized by: Ghidra's auto-analysis (Ghidra Baseline), improvement due to control-flow extraction (CE), and improvement due to feedback-driven refinement (FD).

Experiment: We capture traces for each SUT and process them using Incision. We use two traces for the baseband (Figure 6, left) and six traces for the BCM (Figure 6, right).

Results: For both firmware databases, the use of Incision results in an increase in the number of identified functions over Ghidra. For the baseband, the increases are due to adding task-level CFGs into the database and the subsequent feedback process. The task-level CFGs trigger further discovery of functions not found in either the processed traces or original database. For the BCM, all increases are due to the extracted CFGs. Notably, by combining static and dynamic analyses in a feedback loop, Incision recovers more function starts than present in either the traces or the baseline Ghidra database.

5.2.4 Database Error Correction (C.1.4)

We assess this criterion using the same firmware and traces as the previous section (Section 5.2.3), as shown in Figure 6.

Experiment: We process each trace using Incision and assess how it rectifies reported disassembly errors. For each “error block” our tool reports it can fix automatically, we verify that the proposed correction is in line with how a human analyst would proceed.

Results: For the BCM firmware, we encounter no disassembly errors for any of the traces. For the baseband firmware, we encounter errors for both traces; trace T1 yields 53 errors out of 88k blocks, and T2 yields 188 out of 200k blocks. Our results are in line with our expectations. The BCM firmware uses the V850ES instruction set, which has a single configuration. Therefore if our traces contain correct block starts, then no disassembly errors should be introduced by processing them. In contrast, the baseband firmware uses both the ARM and Thumb2 instruction sets. As reported by Friebertshäuser [Fri], most tools perform relatively poorly for firmware that mixes these instruction sets since the correct disassembly context is often only apparent at runtime.

For trace T1, Incision reports a fix for 33 out of the 53 errors. Of these, only one is incorrect. On investigation, we find that this is because Ghidra reverts Incision's fix. Ghidra performs a "Non-returning Function Detection" pass as part of its *auto-analysis*, which Incision triggers as part of its feedback process. The result is that the thunk function inserted by Incision gets removed by Ghidra. For trace T2, Incision proposes fixes for 135 out of the 188 errors, all of which are correct. Overall, we find that disassembly errors rarely occur. However, when present, Incision can apply automatic fixes that handle the majority of cases.

5.3 Real-World Usability (C.2.1, C.2.2)

Despite the large number of automated firmware analysis techniques proposed in the literature, it is rare to find any evaluated using firmware extracted from end-user devices. In practice, it is often the case that these techniques operate under specific assumptions, which are target-dependent and may not always hold. To apply these techniques to end-user devices, we must almost always perform preliminary manual reverse engineering to make necessary adaptations.

In this section, we show how Incision can complement techniques by supporting that preliminary reverse engineering effort. To that end, we evaluate the firmware of two end-user devices. These devices and their firmware share several properties that make them challenging to analyze. ❶ Both firmware coordinate various complex layers and protocols (RRC for the baseband and an immobilizer [WdHG+20] and diagnostics [dHG18] for the BCM). ❷ Even with JTAG access, acquiring representative execution traces from either device remains a challenging task. Due to a lack of tracing hardware on the BCM, as it is not real-time constrained, we resort to emulation to acquire traces. For the baseband, we reverse engineer and configure the on-chip tracing hardware for trace acquisition. However, our traces are often incomplete and limited in size. Thus, we must carefully select which regions to trace in order to perform practical analysis. ❸ Both devices rely on various on- and off-chip peripherals. The baseband incorporates an RF component for LTE radio communication. While the BCM includes an immobilizer base station, a low-frequency antenna, and peripherals to communicate over automotive buses (e.g., CAN).

5.3.1 Identifying Stall States to Emulate Renault BCM Firmware

In this experiment, we use Incision to aid in locating stall states in the BCM firmware that prevent us from successfully emulating the device.

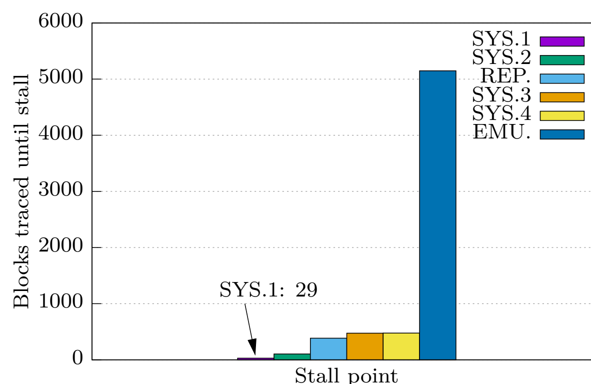

Figure 7: Traced block coverage (unique blocks) for BCM firmware for each stall point; tracing halted when buffer capacity (1MB) reached. After providing stimulus at each stall point (SYS.N, REP.), we are able to emulate the device firmware (EMU).

Set-up: As read/write protection is disabled on the BCM, we can obtain its firmware over JTAG. We use Renesas CubeSuite+ IDE to emulate the firmware and acquire traces.

Objective: We assess the effectiveness of INCISION to aid firmware emulation by detecting stall states in traces. An emulated firmware instance should be as true to the on-target execution as possible. However, emulation often runs into problems where the code hangs on a peripheral status register [FML20]. Thus, we need to handle such states to facilitate emulation. We can identify such states in an execution trace by locating repeated flows that saturate the trace buffer. They might manifest as long repeated flows of code or short tight loops on particular blocks, e.g., when polling a peripheral register.

Experiment: We attempt to emulate the firmware by applying INCISION iteratively. In each iteration, we execute the firmware from the reset vector and halt when the trace buffer (limited to 1MB) is full. Since our objective is emulation, we do not restrict traces by a policy. Instead, we use policies to detect stall states; we define the following policies: ⟨`(tst1 ?, DAT_fffff*)+`⟩, to detect any repeated instructions which poll for a register bit in the memory region where peripherals reside, and ⟨`(cmp r0, r*)+`⟩, to detect if emulation stalls on the return value of a specific function.

Results: Figure 7 shows how INCISION gradually improves emulation over successive traces. The first trace makes little progress and stalls on an instruction that polls a bit in the *SYS* register. Since INCISION can identify the flow responsible for the stall, we can clear the corresponding bit in the register to advance the emulation past this point. Over all iterations, INCISION enables us to identify four stall-states waiting on the *SYS* register and one longer repeated sub-flow, which saturates the trace buffer. In the latter case, the repeated flow consists of two function calls. The emulation only proceeds if the second returns successfully. Even though approaches such as P^2IM [FML20] reportedly handle tight polling loops, such as those on the *SYS* register, it is unclear how they would identify the latter stall state. In our last trace, where we achieve emulation (EMU.), we observe a dramatic increase in the number of unique blocks executed after passing the initial states.

5.3.2 Analysing the Key Usage of the Huawei R216h

In this experiment, we use INCISION to aid in analyzing how the baseband handles sensitive key material.

Set-up: We obtain the baseband firmware over JTAG and configure tracing as detailed in Appendix B. We also use JTAG to obtain RAM dumps. As mentioned previously, for this device, the CoreSight trace capture buffer is limited to 1MB, and the number of regions we can specify as inclusions or exclusions is limited to four. We provide input to SUT using an Ettus USRP B210 Software Defined Radio (SDR). We detail the complete experimental set-up in Appendix C.

Objective: Our analysis goal is to identify how the firmware uses cryptographic keys and check that it sanitizes them correctly. We use INCISION to perform the first stage of analysis: to identify crucial functions and obtain relevant traces. We then use the improved database to help identify the buffers storing key material and analyze how the firmware uses them.

Experiment: To improve our initial database, we first attempt to capture traces of LTE layers known to handle cryptographic keys using the following policies: ⟨"NAS"⟩ and ⟨"RRC"⟩. From this we obtain traces T1 and T2, as shown in Figure 6 (left). These traces improve the quality of (relevant) information in our database (identified global buffers, static data references, resolved control-flow). They also enable us to form bounds to exclude task-switching behavior in subsequent traces. Next, we take traces excluding the task-switching region and the bounds obtained using the following (more

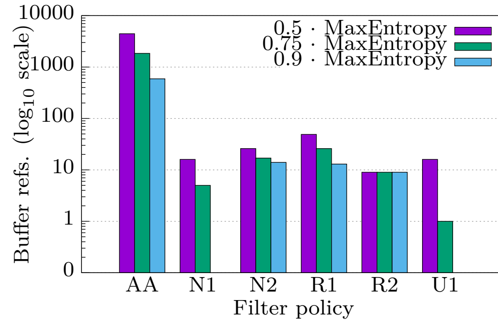

Figure 8: Number of key buffers to analyze. AA: initial database; Nn, Rn, Un: database after processing a trace with references kept if referenced by a block in the trace. We identify viable buffers as those with high entropy. We compute thresholds for keeping buffers as a proportion of their Max. (Shannon) Entropy (ME) relative to their size. Without Incision, max. number of buffers to analyze is 4451 (AA, 0.5·ME), after using Incision, max. number of buffers is 49 (R1, 0.5·ME). We use $\log_{10}$ scale for readability.

specific) policies (traces shown in Figure 8): ⟨"RRC", "RRC_SMC"⟩ (traces R1 and R2), ⟨"NAS", "NAS_EMM", "NAS_LMM"⟩ (traces N1 and N2), and ⟨"USIMM"⟩ (trace U1). Using the extracted sub-flows from the traces captured, we identify firmware regions responsible for each protocol layer. Next, we use those regions and database cross-references to locate global buffers accessed by the code blocks present in identified sub-flows. The key material we are interested in (AES keys) should be at least 16 bytes in size and have high entropy. We use this as a filtering criterion: keeping only references to buffers with these properties that are referenced by code from our identified sub-flows. Finally, we perform a manual analysis of the key usage using the identified buffers and the improved database.

Results: In traces containing task-switching logic (i.e., T1 and T2), Incision correctly identifies its bounds. It identifies the entry-point of `intEnt` as the ISR and a block in `reschedule` as the point responsible for task switching. In the remaining, finer-grained traces, we observe no switching logic as we can exclude its bounds. We find that all of our traces correspond to the policies used to take them. In this way, Incision enables us to perform a reverse engineering task, which requires both static and dynamic analyses with minimal manual effort (see Section 5.4). It integrates information acquired dynamically into the firmware database, which allows us to form finer-grained reverse engineering policies. Further, it enables us to restrict taken traces to contain policy-relevant information by automatically locating the bounds of task switching/handling logic.

When performing the next stage of analysis (to identify key buffers), our improved database and traces enable us to reduce the number of buffers requiring manual analysis. The decrease is by two orders of magnitude—from thousands of buffers to tens, as visualized in Figure 8. Through our analysis, we locate the following keys: k_{nas_i}, k_{nas_e}, k_{enb}, k_{rrc_i}, k_{rrc_e}, k_{up_e}, ck, ik, and k_{asme}. After analyzing the code processing them, we find that, though all identified keys are considered session keys by the standard, only ck, ik, and k_{asme} are correctly sanitized.

5.4 Human Effort (C.3)

We evaluate this criterion using traces T1 and T2 for the baseband firmware. We compute the proportion of errors corrected automatically to those needing manual intervention. For each error requiring manual intervention, we measure its complexity ($\propto$ human effort) by the type and number of changes required to fix it in the database. We consider adding/removing a control-flow edge, label, or function start to be low-effort and performing multiple changes

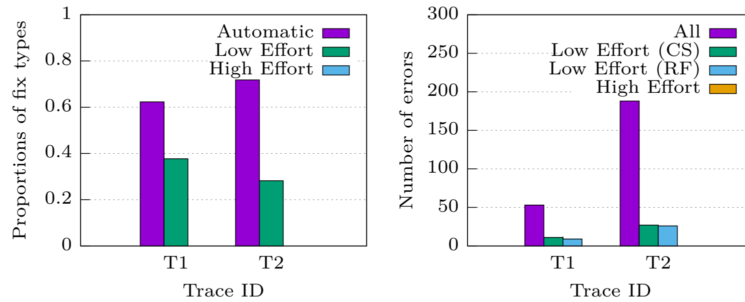

Figure 9: Number of disassembly errors resulting from baseband traces T1 and T2. **Left:** proportion of errors fixed automatically, errors fixed manually that are low effort, errors fixed manually that are high effort. **Right:** breakdown of errors; all errors reported, errors fixed: low effort where we create a function start (CS), where we force Ghidra to re-create a function (RF); high effort.

or other changes as high-effort.

Experiment: Using the same methodology as Section 5.2.4, we manually analyze each un-fixed error. We then apply a suitable fix manually and record the complexity of the fix performed in terms of database changes.

Results: We show the results in Figure 9. For trace T1, we encounter 53 disassembly related errors, and for T2, 188. The majority of errors could be corrected automatically by INCISION. Of those requiring manual intervention, all were "low-effort" fixes, requiring a single change. The majority consisted of creating a missing function start, and the rest, re-creation of a function.

Overall, INCISION dramatically improves the standard reverse engineering workflow for complex device firmware. Although its analyses are not entirely automated, it reduces human effort in almost all cases. Where manual intervention is needed (e.g., to perform tracing), it is unavoidable and not a limitation of our approach. Moreover, the manual effort required to use INCISION is minimal, consisting of minor changes to the database.

5.5 Discussion and Limitations

In this section, we discuss the possible limitations of our approach.

Firmware Composition In a scenario where several tasks within a trace heavily share code, INCISION may incorrectly merge their flows. The may occur if one task resumes to a location where another was preempted (e.g., in library code). In this case, given the limited information available in our traces, there would be no way to distinguish between the flows without additional context.

Reverse Engineering Policy Granularity While a policy can encode realistic target-specific reverse engineering objectives, it cannot encode all possible goals. In particular, a policy cannot express relations between elements. While this could be considered a limitation, we do not consider it a limitation in the context of the tasks INCISION is intended to perform, i.e., the initial reverse engineering effort to enable other analyses.

Soundness and Completeness To handle trace mechanism errors, we estimate viable control-flow to make analysis possible. Therefore, we cannot guarantee soundness or completeness for analyses that build upon INCISION's outputs where such errors are present. We consider this an acceptable trade-off as trace analysis would not otherwise be possible.

Quantification of Human Effort When assessing **C.3**, to avoid biases based on prior human experience, we measure human effort as a function of the number of manual tasks performed and their complexity. This comes at the cost of not directly measuring the time taken to perform tasks (as that would depend on the subject's expertise). We argue that this metric is a reasonable alternative. The time taken to complete tasks is generally proportional to their complexity and amount, irrespective of prior human experience.

Tracing Mechanisms While many end-user devices provide tracing facilities, some vendors remove such features altogether. While it may be possible to alter a device's firmware to add instrumentation to provide a trace mechanism [Sel18], the effort to do so is comparable to many everyday reverse-engineering tasks. Hence, removal of these facilities acts as a countermeasure against performing reverse-engineering using execution traces. Therefore, we consider devices without tracing features out of scope. For other devices, high-end or expensive equipment can overcome some of the problems we describe, such as limited trace buffers. However, for many end-user devices, these tools are often unavailable. We, therefore, exclude analysis scenarios involving them from our evaluation.

6 Related Work

Dynamic firmware analysis AVATAR [MFB18, ZBFB14] and SURROGATES [KKM15] introduced the concept of hardware in the loop analysis. This methodology allows dynamic analysis to be performed without handling many of the complexities of the device being analyzed. It enables hybrid analyses where a fast host can emulate most of the firmware and defer to the device only for I/O and interrupts. While these approaches reduce some of the complexities of analysis, interaction with the underlying device may introduce unacceptable overheads. For example, as AVATAR relies on state transfer between the device and host, the pause introduced during the transfer may lead to the analyzed core being reset. Furthermore, when using this kind of analysis, due to the way devices are interacted with, it is often not possible to achieve reproducible executions. PANDA [DHH+15] addresses this problem by shifting firmware execution to an emulator that can record and replay execution traces. Many approaches attempt to completely rehost the firmware using a generic emulator and provide peripheral models and simulated interrupts to mimic the hardware. To a greater [FML20, MFL+21, CGML20] and lesser extent [GMS+19, HVP+20, CGS+20], techniques can generate these models automatically. Sun et al. [SGZ19] propose a domain-specific reverse engineering framework to extract the high-level semantics of control-logic from Internet of Things (IoT) device firmware. In contrast to INCISION, they require high-quality CFG reconstruction and fine-grained execution traces that include memory loads and stores, making their technique challenging to apply to the devices and firmware analyzed in this work. In order to test firmware for vulnerabilities, many approaches use symbolic execution. Davidson et al. [DMRJ13], for example, demonstrate how for very small firmware, it can be used effectively to achieve complete coverage. Both INCEPTION [CCF18] and HARDSNAP [CF20] focus on how to handle the nuances of complex firmware and devices with multiple peripherals under symbolic execution. They leverage the fact that source code is sometimes available for portions of firmware, which contains significantly more information about high-level constructs and data-types and can thus make symbolic execution more tractable. Other approaches use fuzz testing to assess devices for vulnerabilities. A naive approach to this has significant drawbacks, as highlighted by Muench et al. [MSK+18]. Therefore, for most approaches, the firmware is first manually reverse engineered, and then select parts are fuzzed under an emulated environment [MSP20, RCGH20, SPL+19, ZDY+19]. Aside from academic work, both SYSTEMVIEW [SEG] and TRACEALYZER [Per] support developers in debugging their firmware through trace analysis. However, they require access to the source code for the

firmware, so that it can be compiled with software-based instrumentation. As such, the use of these tools for reverse engineering or analysis of end-user devices is limited.

Static Firmware Analysis While the majority of focus has been on dynamic techniques, a small number of approaches have used static analysis. However, almost all of these have not been applied to bare-metal or monolithic firmware. Costin et al. [CZFB14] perform a large-scale assessment of downloadable device firmware (mostly Linux-based) and apply simple static analyses to discover several vulnerabilities. Shoshitaishvili et al. [SWH+15] demonstrate how symbolic execution with human guidance can be used to discover authentication bypass vulnerabilities in a various firmware, including binary blob firmware. Cojocar et al. [CZV+15] use static analysis and machine learning to locate parsing routines in device firmware automatically. Similarly, Thomas et al. [TCG17, TGC17], use a static approach to locate undocumented functionality in Linux-based firmware.

Control-Flow Recovery and Disassembly A significant body of research has contributed to the state-of-the-art in binary analysis. Much of this work has been towards improving control-flow recovery and disassembly. We refer the reader to the work of Shoshitaishvili et al. [SWS+16] for a more complete exposition. We summarize the contributions most relevant to our work in what follows. IDA Pro [Hex] and Ghidra [Nat] provide solutions to aid manual reverse engineering. They perform function start identification, disassembly, control-flow recovery, and cross-referencing. However, for complex firmware the quality of their analyses is highly dependent on human intervention. Andriesse et al. [ACvdV+16] provide an in-depth analysis of the problems faced when performing disassembly on real-world x86/x64 binaries. They attempt to remedy the situation with a compiler agnostic approach to control-flow recovery [ASB17]. Muhui et al. [JZL+20] perform a similar analysis for ARM-based binaries. The authors of Polypyus [Fri] propose an approach leveraging past reverse engineering efforts on firmware from the same vendor to aid in control-flow and function start recovery. Their tool provides a step forward in addressing a problem which dramatically affects the correctness of disassembly for VLE instruction sets such as Thumb2.

7 Conclusion

Complex embedded devices present many challenges when performing a security analysis: undocumented peripherals, uncommon instruction sets, and limited tool support. Performing any kind of automated analysis, in particular, generally requires a non-trivial amount of upfront manual reverse engineering. In this paper, we present Incision, a framework that reduces the manual effort required to perform these preliminary reverse engineering tasks. We design and implement four novel approaches that automate aspects of the typical reverse engineering workflow: firmware region inference, OS primitive identification, task-aware control-flow recovery from system-wide execution traces, and feedback-driven refinement. The latter combines static and dynamic control-flow recovery to improve our overall understanding of the firmware.

We demonstrate the effectiveness of Incision on the firmware of two end-user devices, a VxWorks-based LTE baseband and an automotive BCM, as well as a set of firmware based on widely used RTOS: FreeRTOS and Zephyr. We demonstrate Incision's effectiveness by performing real reverse engineering tasks. We use it to identify stall-points that prevent emulation and assess cryptographic key usage. Both tasks greatly benefit from the use of Incision. Further, we show that Incision does not introduce significant complexity to the standard reverse engineering process and requires little manual effort to use. Incision's analyses produce correct results with high confidence and are robust to different OSes, ISAs, and trace mechanisms.

References

[ACvdV+16] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 583–600. USENIX Association, 2016.

[ARMa] ARM. CoreSight Program Flow Trace. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0035b/IHI0035B_cs_pft_v1_1_architecture_spec.pdf. Accessed: 2020/06/29.

[ARMb] ARM. CoreSight Trace Memory Controller. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0461b/DDI0461B_tmc_r0p1_trm.pdf. Accessed: 2020/06/29.

[ARMc] ARM. Embedded trace buffer technical reference manual. https://developer.arm.com/documentation/ddi0242/b/. Accessed: 2020/11/05.

[ASB17] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 177–189. IEEE, 2017.

[CCF18] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 309–326. USENIX Association, 2018.

[CF20] Nassim Corteggiani and Aurélien Francillon. Hardsnap: Leveraging hardware snapshotting for embedded systems security testing. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 294–305. IEEE, 2020.

[CGML20] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*, 2020.

[CGS+20] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1201–1218. USENIX Association, 2020.

[CKB17] Lucian Cojocar, Taddeus Kroes, and Herbert Bos. JTR: A binary solution for switch-case recovery. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*, volume 10379 of *Lecture Notes in Computer Science*, pages 177–195. Springer, 2017.

[CZFB14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 95–110. USENIX Association, 2014.

[CZV+15] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. PIE: parser identification in embedded systems. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 251–260. ACM, 2015.

[DBXP20] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1497–1511. IEEE, 2020.

[dHG18] Jan Van den Herrewegen and Flavio D. Garcia. Beneath the bonnet: A breakdown of diagnostic security. In Javier López, Jianying Zhou, and Miguel Soriano, editors, *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, volume 11098 of *Lecture Notes in Computer Science*, pages 305–324. Springer, 2018.

[DHH+15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In Jeffrey Todd McDonald, Mila Dalla Preda, and Natalia Stakhanova, editors, *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC, Los Angeles, CA, USA, December 8, 2015*, pages 4:1–4:11. ACM, 2015.

[DMRJ13] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 463–478. USENIX Association, 2013.

[FML] Bo Feng, Alejandro Mera, and Long Lu. P^{2}IM documentation. http://archive.is/ydJ6x. Accessed: 2020/07/30.

[FML20] Bo Feng, Alejandro Mera, and Long Lu. P^{2}IM: scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1237–1254. USENIX Association, 2020.

[Fri] Jan Friebertshäuser. Polypyus – The Firmware Historian. https://github.com/seemoo-lab/polypyus/. Accessed: 2020/07/25.

[GMS+19] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, pages 135–150. USENIX Association, 2019.

[Hex] Hex-Rays. IDA Pro. https://www.hex-rays.com/products/ida/. Accessed: 2020/07/27.

[HVP+20] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: enabling dynamic analysis of real-world trustzone software using emulation. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 789–806. USENIX Association, 2020.

[JZL+20] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. An empirical study on ARM disassembly tools. In Sarfraz Khurshid and Corina S. Pasareanu, editors, *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 401–414. ACM, 2020.

[KKM15] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: enabling near-real-time dynamic analyses of embedded systems. In Aurélien Francillon and Thomas Ptacek, editors, *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015*. USENIX Association, 2015.

[MFB18] Marius Muench, Aurélien Francillon, and Davide Balzarotti. Avatar^2: A multi-target orchestration platform. In *BAR 2018, Workshop on Binary Analysis Research, colocated with NDSS Symposium, 18 February 2018, San Diego, USA*, San Diego, 2018.

[MFL+21] Alejandro Mera, Bo Feng, Long Lu, Engin Kirda, and William Robertson. Dice: Automatic emulation of dma input channels for dynamic firmware analysis. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, 2021.

[MSK+18] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[MSP20] Dominik Maier, Lukas Seidel, and Shinjo Park. Basesafe: baseband sanitized fuzzing through emulation. In René Mayrhofer and Michael Roland, editors, *WiSec '20: 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Linz, Austria, July 8-10, 2020*, pages 122–132. ACM, 2020.

[Nat] National Security Agency (NSA). Ghidra. https://ghidra-sre.org/. Accessed: 2020/07/27.

[Per] Percepio. Tracealyzer. https://percepio.com/tracealyzer/. Accessed: 2020/07/25.

[RCGH20] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 19–36. USENIX Association, 2020.

[Ren] Renesas. CubeSuite+ (CS+) . https://www.renesas.com/us/en/products/software-tools/tools/ide/csplus.html. Accessed: 2020/07/31.

[SEG] SEGGER. SystemView. https://www.segger.com/products/development-tools/systemview/. Accessed: 2020/07/25.

[Sel18] Denis Selianin. Researching Marvell Avastar Wi-Fi: From zero Knowledge to Over-the-air zero-touch RCE. https://2018.zeronights.ru/wp-content/uploads/materials/19-Researching-Marvell-Avastar-Wi-Fi.pdf, 2018. Accessed: 2020/07/30.

[SGZ19] Pengfei Sun, Luis Garcia, and Saman A. Zonouz. Tell me more than just assembly! reversing cyber-physical execution semantics of embedded iot controller software binaries. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 349–361. IEEE, 2019.

[SPL+19] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard E. Shrobe, and Mathias Payer. Firmfuzz: Automated iot firmware introspection and analysis. In Peng Liu and Yuqing Zhang, editors, *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, IoT S&P@CCS 2019, London, UK, November 15, 2019*, pages 15–21. ACM, 2019.

[SWH+15] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157. IEEE Computer Society, 2016.

[TCG17] Sam L. Thomas, Tom Chothia, and Flavio D. Garcia. Stringer: Measuring the importance of static data comparisons to detect backdoors and undocumented functionality. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, volume 10493 of *Lecture Notes in Computer Science*, pages 513–531. Springer, 2017.

[TGC17] Sam L. Thomas, Flavio D. Garcia, and Tom Chothia. Humidify: A tool for hidden functionality detection in firmware. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, volume 10327 of *Lecture Notes in Computer Science*, pages 279–300. Springer, 2017.

[War63] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[WdHG+20] Lennert Wouters, Jan Van den Herrewegen, Flavio D. Garcia, David F. Oswald, Benedikt Gierlichs, and Bart Preneel. Dismantling dst80-based immobiliser systems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):99–127, 2020.

[ZBFB14] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[ZDY+19] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1099–1114. USENIX Association, 2019.

A Algorithms

Algorithm 1: Region inference.

In: D: program database, P: policy, R: max. number of regions, δ: grouping distance.

Out: M_r: at most R region bounds for P.

```
1  Rs ← staticAndSymbolRefs(D), M ← {}
2  foreach r ∈ Rs do // region identification
3  |   if match(D, P, r) then
4  |   |   k ← matched(D, P, r)
5  |   |   f ← enclosingFn(D, r)
6  |   |   M_k ← M_k ∪ {(r, f)}
7  M_r ← rankFnRefs(D, M, δ)
8  if |M_r| > R then // region grouping
9  |   M_r ← agglCluster(D, M_r, R)
10 return M_r
```

Algorithm 2: Transition classification.

In: D: program database, $G^{\rightarrow}$: dynamic ICFG, b_i: first block, b_j: successor of b_i, S: call stack, N: reachability bound, T: execution trace.
Out: C: classification, $G^{\rightarrow}$: new dynamic ICFG, S: new call stack.

1 $G \leftarrow$ *ICFG*(D), $C \leftarrow$ *unresolved*
2 **if** *disasmError*(D, b_i) $\vee$ *disasmError*(D, b_j) **then**
3 | $C \leftarrow$ *error*
4 **else if** *traceError*(T, b_i, b_j) **then**
5 | **if** *reachableIn*($G^{\rightarrow}$, b_i, b_j, S, N)
6 | $\vee$ *reachableIn*(G, b_i, b_j, S, N) **then**
7 | | $C \leftarrow$ *viable*
8 **else if** *directFlow*(G, b_i, b_j, S) **then**
9 | $C \leftarrow$ *viable*
10 **else if** *isIndirectFlowOut*(G, b_i) **then**
11 | $f \leftarrow$ *enclosingFn*(D, b_i)
12 | $f_{b_i} \leftarrow$ *propagateConstsAndBounds*(D, f)
13 | **if** *indirectFlow*(G, f_{b_i}, b_i, b_j, S) **then**
14 | | $C \leftarrow$ *viable*
15 **if** $C \neq$ *error* $\wedge \neg$*traceError*(T, b_i, b_j) **then**
16 | $G^{\rightarrow} \leftarrow$ *updateDynamicICFG*($G^{\rightarrow}$, b_i, b_j)
17 **else if** $C =$ *viable* **then**
18 | $S \leftarrow$ *updateCallStack*(G, b_i, b_j, S)
19 **return** $\langle C, G^{\rightarrow}, S \rangle$

Algorithm 3: Suffix and LCP array construction.

In: T: execution trace, d: prefixes or suffixes.
Out: S: suffix array, L: LCP array w/o overlaps, M_T: address to suffix array mapping.

1 $\Sigma \leftarrow$ *makeAlphabet*()
2 **foreach** $b \in T$ **do**
3 | **if** *address*(b) $\notin \Sigma \vee$ *hasError*(b) **then**
4 | | $k \leftarrow$ *next*(Σ)
5 | | $\Sigma_b \leftarrow k$
6 $\Sigma^{\perp} \leftarrow$ *next*(Σ) `// compute sentinel`
7 **if** *computeBackwards*(d) **then**
8 | $M_T \leftarrow$ *makeReverseMapping*(T, Σ, $\Sigma^{\perp}$)
9 **else**
10 | $M_T \leftarrow$ *makeMapping*(T, Σ, $\Sigma^{\perp}$)
11 $S \leftarrow$ *SuffixArray*(M_T, $\Sigma^{\perp}$)
12 $L \leftarrow$ *removeOverlaps*(*LCPArray*(M_T, S))
13 **return** $\langle S, L, M_T \rangle$

Algorithm 4: Task switch start identification.

```
In: D: program database, L_p: LCP array for prefixes, C_u: unresolved transitions,
    N: min. prefix length, k: max. ISRs, δ: min. proportion of satisfying
    transitions.
Out: C: transitions into the ISR.
1  G ← ICFG(D)
2  M ← {}, P ← {}, T ← {}
3  foreach (b_i, b_j) ∈ C_u do
4  |  f ← enclosingFn(D, b_j)
5  |  if isFnStart(D, b_j) ∧ ¬reachable(D, b_i, f) then
6  |  |  M_{b_j} ← succ(M_{b_j})
7  |  T_{b_j} ← succ(T_{b_j})
8  foreach b_j ∈ M do
9  |  if M_{b_j}/T_{b_j} > δ then
10 |  |  v ← repeatedPrefixes(L_p, N, b_j)
11 |  |  if |v| = 1 then
12 |  |  |  P_{b_j} ← v
13 C ← take(k, sort(P))
14 return C
```

Algorithm 5: Control-flow extraction.

```
In: D: program database, F: identified flows, C: transition classifications, P:
    policy, L: LCP array, η: merge threshold.
Out: F→: flows grouped by tasks, G⇉: map of tasks to dynamic CFGs with
     policy labels.
1  F→ ← {}, G⇉ ← {}, prev ← ⊥, pKind ← ⊥
2  foreach f ∈ F do
3  |  b_s ← startBlock(D, f), b_e ← endBlock(D, f)
4  |  if ¬isTask(pKind)
5  |  ∧ isBreaks(C, endBlock(D, prev))
6  |  ∧ isBreaks(C, b_s) then
   |  |  // contiguous switching logic
7  |  |  prev ← mergeFlow(prev, f)
8  |  else if isBreaks(C, b_s) ∧ isBreaks(C, b_e) then
9  |  |  updateFlow(F→, G⇉, L, P, prev, pKind)
10 |  |  prev ← f, pKind ← context-switch
11 |  else
12 |  |  updateFlow(F→, G⇉, L, P, prev, pKind)
13 |  |  f′ ← findMatchingFlow(F→, f, η)
14 |  |  if f′ ≠ ⊥ then
15 |  |  |  pKind ← task-resume
16 |  |  else
17 |  |  |  pKind ← task-start
18 |  |  prev ← f
19 updateFlow(F→, G⇉, L, P, prev, pKind)
20 return ⟨F→, G⇉⟩
```

Algorithm 6: Feedback-driven refinement.

In: D: program database, $G^{\rightrightarrows}$: map of tasks to dynamic CFGs, C_e: error transitions.
Out: D: new program database.

```
1 foreach G ∈ CFGS(G⇉) do
2 |   D ← mergeDatabaseCFG(D, G)
3 foreach e ∈ C_e do
4 |   blocks ← disassembleFrom(D, e)
5 |   if blocks = ⊥ then
6 |   |   blocks ← updateViaOracle(D, e)
7 |   D ← updateDatabase(D, blocks)
8 D ← autoAnalyse(D)
9 return D
```

B Huawei R216h 4G/LTE CoreSight Configuration

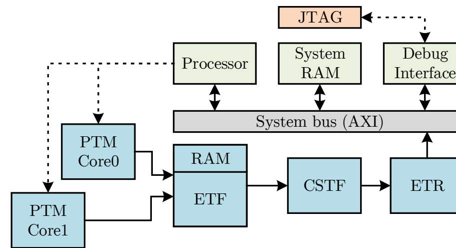

Figure 10: CoreSight configuration as used on the Huawei R216h 4G/LTE modem. Trace data from the two PTMs which monitor cores 0 and 1 of the ARM CPU is initially stored in the local memory of the Embedded Trace FIFO (ETF) module. From there trace data is fetched via the CoreSight Trace Funnel (CSTF) by the Embedded Trace Router (ETR) module and transferred to the system RAM. Finally, the trace data in RAM is accessed externally through the JTAG interface.

The HiSilicon ARM processor on the Huawei R216h 4G/LTE baseband has tracing capabilities provided by ARM CoreSight [ARMb]. We can configure and access the tracing peripherals through the JTAG interface. Figure 10 depicts the configuration for our device. In our setup, we configure the trace data to flow from the Program Trace Module (PTM), which traces the VxWorks core, through the Embedded Trace Funnel (ETF) and CoreSight Trace Funnel (CSTF), to the Embedded Trace Router (ETR), which finally writes the data to a buffer we specify in the RAM. We can then halt the device and access the trace data in the buffer through JTAG. The hardware also contains a Trace Port Interface Unit (TPIU), which outputs the trace as a stream. However, since we did not know the pin-out for this trace port, and capturing the stream would require specialized (and expensive) hardware, we did not attempt to trace the device in this way.

Our setup facilitates two types of traces: ❶ *complete traces* which contain trace data for all the code blocks starting from a specific given start address to either a given stop address or to when the trace buffer is exhausted, and ❷ *address range traces* which contain trace data for code blocks that fall within a specified range, similarly capped to the size of the storage space. A combination of ❶ and ❷ is also possible where ❷ is used as an exclusion range such that trace data can be obtained for everything that is not contained inside the excluded range(s).

C Analyzing the Huawei R216h Baseband with Incision

In this section, we provide a break-down of the steps taken to reverse engineer the Huawei R216h baseband firmware using INCISION. We visualize the analysis set-up and data-flow in Figure 11, which expands on Figure 1 to show the additional manual intervention required to use INCISION with this particular device.

As described in Section 5.3.2, the hardware set-up consists of an Ettus USRP B210 SDR (SDR in Figure 11), the Huawei R216h configured as detailed in Appendix B (SUT in Figure 11), and a laptop with Ghidra and INCISION installed. The analysis process is orchestrated by a human-in-the-loop, depicted by the purple face in Figure 11. As in the *standard* workflow to reverse engineer devices of this kind, we proceed by obtaining a dump of the device firmware and bootstrap the reverse engineering process by loading the dump into Ghidra and triggering its auto-analysis pass (step ❶ in Figure 11). Following

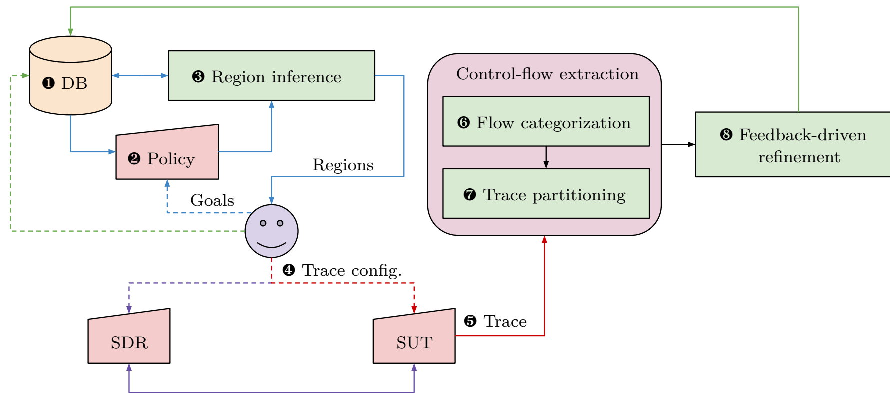

Figure 11: Overview of Incision applied to analyze the Huawei R216h baseband firmware. We indicate the human-in-the-loop by a purple face. We depict data-flow by block arrows where propagation is automated by Incision, and dashed arrows where manual intervention is required.

this set-up stage, we proceed to reverse engineer the device firmware by applying Incision in feedback loops.

In the next step, we form an initial reverse engineering policy based on our goal to analyze how the device handles cryptographic keys (step ❷). We encode this policy using a simple Python-based API provided by Incision, and use it as input to the region inference algorithm (step ❸). As the device's trace hardware is limited in the number of regions that can be traced, we supply the algorithm with an upper bound on the number of regions to output (a value of 4). We depict the data-flow of this stage in blue.

The result of applying region inference is a set of region bounds, which we use to configure the CoreSight-based tracing mechanism on the device (step ❹). Since the analysis goal is to capture traces where the device performs operations using cryptographic keys, we configure the SDR to provide suitable triggering inputs to the device (data-flow depicted in purple). Following this, we perform trace capture (data-flow in red), which we terminate upon observing sufficient I/O interaction between the SDR and device. This provides us with a trace (step ❺), which we use as input to Incision's control-flow extraction procedures (steps ❻ and ❼).

The final step in the loop (step ❽) integrates the extracted control-flow information back into the firmware database (data-flow in green). For the first two loop iterations, this process requires manual intervention, as described in Section 5.4. We continue to apply Incision, as outlined in Section 5.3.2 to capture additional traces which further refine the firmware database and provide a basis for analysis of the key usage in the firmware.