CAUID: Chip Against Unintended Information Disclosure

Bachelor Semester Project S5 (Academic Year 2025/26), University of Luxembourg

Bernard Steenis
bernard.steenis@uni.lu
University of Luxembourg
Esch-sur-Alzette, Luxembourg

Georgi Bozhkov
georgi.bozhkov.001@student.uni.lu
University of Luxembourg
Esch-sur-Alzette, Luxembourg

Abstract

Contemporary malware typically targets software security systems running in the same execution environment as the protected system. This inherently introduces asymmetry: while defences rely on software components that can be disabled, bypassed, or misled by well-crafted attacks, those components are under the control of the attacker. CAUID (Chip Against Unintended Information Disclosure) investigates a hardware-orientated approach to attack detection that runs independently of the traditional software stack, aiming to minimise dependence on the integrity of the operating system and reduce the trusted computing base.

This BSP further develops CAUID as a research system idea for signature-based monitoring. The scientific deliverable starts with an explanation of what malware is and what a "malware signature" is, and then proceeds to examine a research question concerning multi-pattern matching and variable-length signatures. The technical deliverable shows the improvements implemented that allow CAUID to be more flexible and configurable.

Introduction

The security system is generally implemented at the same privilege level as the system it is intended to protect. Although the popular choice of using a software-based system for an antivirus solution is common, it is also important to note that this form of system is dependent on the reliability of the operating system, the drivers, and the update process. This is a weakness because the malware could target this.

Hardware monitoring provides a different point of view. By moving aspects of detection and enforcement out of the typical software execution environment, a system can decrease the size of the trusted computing base and make certain types of attacks harder. CAUID is an example of this line of work as a research prototype that seeks to identify malicious behavior via signature matching of observed execution traces. Conceptually, these approaches can be lightweight and deterministic, but they also have real-world issues, especially when the size of the signature set increases and when signatures are not of uniform size.

In order to address this from a scientific point of view, the scientific part of this BSP begins by explaining what exactly the term "malware" represents, as well as the term "malware signature," and the reason for the context-dependent nature of signatures. The research question is formulated as follows: **Does supporting many signature lengths make hardware signature matching inefficient?** Previous works on the topic of multi-pattern matching are also discussed in order for the question to be answered in the concluding part of the scientific part of this BSP.

The technical section of this BSP is concerned with the practicality of the security chip that has been implemented. It describes the improvements made to CAUID in order to accommodate signatures of different sizes. The technical section is divided into Improvements, Code explanation, and Assessment, where the latter section is a summary of the work done, as well as a look into the future.

The remainder of this project first introduces the required background knowledge and development environment and states the project tasks and goals.

Project Description

This BSP is structured into four main parts:

- **1. Foundational Requirements** — Presents the essential background knowledge assumed for this BSP and lists the equipment and toolchain used for development and testing.
- **2. Project Tasks** — States the concrete objectives of this BSP, including the scientific and technical deliverables.
- **3. Scientific Deliverable** — The scientific part of the project starts with a brief subsection on malware and malware signatures, which explains the motivation for the use of syntactic or context-dependent signatures. It is followed by the presentation of the research question **"Does supporting many signature lengths make hardware signature matching inefficient?"** and the analysis of the previous work on multi-pattern matching and variable-length signatures. The scientific part concludes with a concluding subsection, where the answer to the research question is provided.
- **4. Technical Deliverable** — The technical deliverable explains and shows the implemented improvements that increase the adaptability and configurability of the security chip. including support for signatures of different sizes and a controlled configuration and inspection workflow. The technical section is organized into Improvements, Code explanation, and Assessment.

1 Foundational Requirements

Knowledge

The project assumes familiarity with the following areas:

- **Hardware description languages (HDL):** ability to read and reason about modular VHDL designs and interfaces.
- **Computer architecture basics:** instruction execution, control/data paths, interrupts/context switching at a conceptual level, and basic micro-architectural observability.

Equipment

Development and experimentation were conducted using:

- **FPGA development board:** Digilent Basys 3 (used for user I/O through switches, buttons, LEDs, and on-board display resources).
- **Toolchain:** FPGA synthesis and implementation tools (Vivado) and simulation for functional validation.
- **Environment:** a simple CPU/SoC setup used to integrate CAUID as an external monitoring component and to exercise representative execution traces.

2 Project Tasks

This BSP is structured around two tasks, corresponding to the scientific and technical deliverables.

Task 1: Scientific deliverable

In this section, the objective is to find an answer to the question: **"Does supporting many signature lengths make hardware signature matching inefficient?"** To achieve this, the scientific deliverable will first introduce what malware is and what the term "malware signature" represents, including the context in which signatures can be used, as they are not necessarily only composed of raw code. The scientific part will also address the concept of multi-pattern matching and the handling of signatures of different lengths, including what the most important factors are in terms of scalability and what the significant bottlenecks are, as they have been reported in the literature. The section will conclude with the addressing of the question that was formulated.

Task 2: Technical deliverable

The second task is to extend CAUID as a reusable hardware security component with increased adaptability and configurability. This will enable it to be integrated into a wider range of CPU-based systems while ensuring that there is a clear module boundary. Furthermore, the technical deliverable provides information on how the chip is able to handle signatures of varying sizes rather than a fixed signature length ones, thus allowing the system to be more flexible and represent a wide range of signature patterns. It also provides information on the configuration and inspection process that was followed during the development process.

The technical section is divided into three sub-sections. In the section on "Improvements", we present what has been newly developed with respect to the previous version of CAUID. This section also includes an explanation of why such improvements have led to better portability, safety, and usability. In "Code Explanation", we present an explanation of the code with the help of code snippets. This explanation is based on the workflow from configuration input and visual feedback to runtime matching and detection. In the "Assessment", we present an explanation of what has been achieved with this BSP and possible directions for future versions.

3 Does supporting many signature lengths make hardware signature matching inefficient?

Malware and malware signatures

In the security literature, malware is defined as "software with a malicious intent that can cause harm to a system or its users, usually by compromising confidentiality, integrity, or availability." [7] Under this general term, different types of malware are classified, such as viruses, worms, and Trojan horses, but it is not the particular form of the malware that characterises it, but the harmful effect it causes intentionally, as well as the attacker's objectives. [7]

A signature is a recognizable pattern used to identify known attacks or attack classes, and signature-based detection is widely used in intrusion detection and prevention systems. [8] In the simplest case, a signature is a distinctive byte sequence (or instruction sequence) that appears in a malware sample or exploit payload. However, purely syntactic signatures are fragile: malware authors can apply transformations that alter the low-level byte or instruction representation while preserving behavior, thereby evading detectors that rely only on surface form. [9]

For this reason, many works treat a malware signature as more general than a fixed substring and consider *semantics-based* characterization, where the aim is to capture behavior (what the program does) rather than the exact code (how it is written). Dalla Preda et al. argue that program semantics provides a formal model of behavior and propose a framework to reason about robustness of malware detectors under classes of obfuscations, using trace semantics and abstraction to ignore irrelevant syntactic variation. [9] Similarly, Feng et al. explicitly distinguish between syntactic signatures (e.g., instruction sequences) and semantic signatures (e.g., control-flow or data-flow properties), and synthesize signatures that describe shared malicious functionality in terms of semantic metadata and relationships between components. [10]

This also clarifies why a signature can depend on context. In networked settings, signatures are often defined over payload and protocol structure (e.g., fields and sequencing constraints), because the same byte pattern may be benign or malicious depending on where and how it appears. [8] In host-based settings, context can include execution traces, control-flow structure, and data-flow properties, where the goal is to identify a malicious computation even if its byte-level encoding changes. [9, 10] Consequently, when a system claims to perform "signature matching," the scientific question is not only *how* matches are computed, but also what kind of pattern is being matched and which contextual signals are part of the signature definition. [8, 10]

Scientific Question

Signature-based detection can be modeled as multi-pattern exact matching: given a stream of symbols (e.g., bytes), the system must determine whether any of a large set of signatures appears as a contiguous substring. In realistic rule sets, signatures are not uniform in size; instead, they span a wide range of lengths, from short patterns to significantly longer ones [4]. This length diversity is not a minor formatting issue; it changes what it means for a design to be scalable because it interacts directly with memory organization, throughput constraints, and the amount of work required to confirm matches [3, 4]. Empirical IDS studies and systems papers motivate this concern by showing that matching can dominate software processing costs and that scaling to large, diverse rule sets becomes a bottleneck [3].

A first major family of solutions uses deterministic automata for exact matching, most notably Aho–Corasick (AC) style constructions [2]. The scientific appeal of AC is that it provides predictable per-symbol processing: the stream is processed in a single pass and matching time is linear in the input length [2]. The practical drawback, emphasized in IDS-oriented research, is that AC-style automata representations can become memory-hungry as rule sets grow [3, 5, 6]. When signatures have many different lengths and share relatively little structure, the number of states and transitions that must be represented can expand substantially, translating into memory pressure (tables, pointers, transition encodings) on the hardware side [3, 5, 6]. This is why several works focus on compressing or restructuring AC-related representations: the dominant limiter is frequently memory footprint rather than raw computation [3, 5, 6].

A second family of solutions approaches the problem differently: instead of committing to a fully deterministic exact-matching structure for the entire rule set, it uses a compact *prefilter* to reject the majority of non-matching substrings quickly and performs exact verification only for the remaining candidates [1, 4]. Bloom filters provide the theoretical foundation for this idea by enabling a controlled space–error trade-off: membership queries are memory-efficient and fast, but can return false positives at a rate determined by the filter parameters [1]. For signature matching, Bloom-filter-style prefiltering is scientifically relevant because it shifts the design problem from “store everything exactly” to “store a compact summary and verify selectively” [1, 4].

Length diversity becomes a central issue again in prefilter-based designs. A straightforward method would maintain separate filtering structures for each signature length, but real rule sets are unevenly distributed across lengths, making “one structure per length” inefficient [4]. The Extended Bloom Filter (EBF) approach addresses this by augmenting filter buckets with additional metadata so that candidate verification remains small, and by reducing the need to maintain a separate structure for every possible length [4]. The EBF results show that, with suitable parameters, false positive rates can be made low and the average verification work can remain small, supporting the claim that scalability can be achieved without storing a large exact structure for all lengths simultaneously [4].

Handling *long* signatures highlights why variable-length support is non-trivial in hardware. Long patterns are costly if matched naïvely at full length because they increase storage and may require deeper verification pipelines [4]. A common strategy is to introduce a threshold length and represent long signatures via bounded-length segments, with bookkeeping to confirm that segments occur in the correct alignment and order [4]. This reframes “arbitrary-length matching” into bounded-length matching plus verification state, reducing the number of lengths the fast front-end must support directly [4].

Across these approaches, the scientific picture is consistent: variable-length signatures do not merely increase the number of stored patterns; they increase structural diversity, which stresses memory and complicates efficient matching if handled naïvely [3–6]. Deterministic automata-based designs respond by compressing transitions and restructuring state representation to fit within feasible memory budgets [3, 5, 6]. Prefilter-and-verify designs respond by limiting the work done per input symbol and accepting a controlled verification stage grounded in Bloom filter theory [1, 4]. Later work targeting large rule sets emphasizes that scalability is achieved by controlling memory growth and keeping per-symbol processing bounded, rather than treating every length independently [3, 6].

Scientific Conclusion

Yes, supporting many signature lengths *can* make hardware signature matching inefficient, primarily because length diversity amplifies memory requirements and design complexity when represented directly. [3–6]. The literature also shows that this inefficiency is not unavoidable: it can be mitigated by compressing and restructuring deterministic automata to reduce memory growth [3, 5, 6], and/or by using compact prefiltering with targeted verification (including segmentation-based methods that support long signatures through bounded-length matching plus lightweight validation) [1, 4]. In other words, signature-length diversity is a key driver of the resource–throughput trade-off, and scalable hardware solutions are those that control the memory impact of that diversity while preserving correct matching [3, 4, 6].

4 Technical Implementation Improvements

The technical improvements made to CAUID during this iteration of the BSP enhance the functionality of the security chip and reduce the gap between the conceptual working principle and a believable real product. Compared to the previous CAUID iteration, this BSP places stronger emphasis on safe configuration, clearer user interaction under strict Basys3 I/O constraints, and a more portable integration style.

- **Custom signatures support** In the previous iteration, signature content was primarily treated as predefined or only minimally changeable during development, which limited how realistically the system could be exercised with new threat patterns. In this BSP iteration, CAUID supports interactive signature authoring on the target board, enabling signatures to be entered, saved, and later reloaded for inspection. This is closer to the operational reality of signature-based systems, where detection rules evolve over time and must be iterated during testing, tuning, and validation.

- **Additional visible feedback for the entered input.** Earlier interaction patterns provided limited visibility into what was currently being entered or selected, which increased the likelihood of operator error and slowed down debugging. This BSP introduces explicit visual feedback that reflects the current input state (for example: showing the current nibble input, selected rule index, and progress through an input sequence). This makes configuration safer because incorrect entries become immediately visible, and it makes development more efficient because the user can verify the input stream without relying on external tooling.

- **Variable signature length** The previous iteration relied on a fixed-length matching assumption, which is simple but does not reflect how signature sets behave in practice. This BSP improves configurability by allowing signatures to be treated as variable-size at the level of comparison logic. Concretely, signatures remain stored in a fixed 32-bit representation for hardware simplicity, but a per-rule byte-enable mask determines which bytes are meaningful for matching, and a mask of 0000 disables the rule entirely. This approach is more robust than a naive variable-length compare because it preserves deterministic, synthesis-friendly hardware behavior while still allowing short signatures and partially specified patterns when needed.

- **CAUID logic code** A key portability improvement in this iteration is the separation between system integration and CAUID behavior. Previously, parts of the configuration or control logic of the security chip were located inside the top-level of the CPU, making the design incompatable with non-adjusted processers and harder to reason about as a standalone component. In this BSP, configuration and inspection behavior is implemented inside the CAUID module, while the CPU top layer remains limited to interconnection and output multiplexing. This matches the expected structure of a reusable security system where the host system provides signals and display routing, while CAUID encapsulates its own internal state, safety gating, and user interaction logic.

Code explanation

This part explains the implementation through selected code snippets. The goal is to provide a clear, step-by-step description of how configuration actions (editing and inspection) are processed, how these actions map to visual feedback, and how runtime observation and mask-based matching lead to detection. The customization features in this BSP (editing, saving, and inspecting signatures) are intended to model the update mechanism of a real hardware security product: instead of being compiled-in constants, rules can be entered, validated, and audited through a controlled workflow.

Inputs and user controls (buttons and switches). The Basys3 buttons are debounced by the `interface` module and provided to `cauid` as a 4-bit bus `buttons`. Internally, `cauid` triggers actions only on rising edges (`0`→`1`) using `buttons_prev`, so holding a button does not repeatedly trigger operations (Figure 1). The button mapping is:

- `buttons(3)` = `btnU`: append one input nibble (edit mode) or load selected rule for inspection (inspect mode)
- `buttons(2)` = `btnL`: clear the current edit buffer (edit mode only)
- `buttons(1)` = `btnR`: save the current edited signature and mask to the selected slot (edit mode only)

```
btnu_edge := buttons(3) and (not buttons_prev(3));
btnl_edge := buttons(2) and (not buttons_prev(2));
btnr_edge := buttons(1) and (not buttons_prev(1));
btnd_edge := buttons(0) and (not buttons_prev(0));

buttons_prev <= buttons;
```

Figure 1: Edge-triggered button handling using `buttons_prev` in `cauid.vhd`.

```
digits_mux <= ui_digits when (cpu_state = x"0") else digits;
```

Figure 2: 7-seg source muxing in configuration vs runtime from `cpu_top.vhd`.

```
if (state = x"0") and (mode_sel = '1') then
    if page_sel = '1' then
        ui_digits_int <= display_reg(15 downto 0);
    else
        ui_digits_int <= display_reg(31 downto 16);
    end if;
```

Figure 3: 7-seg feedback logic in `cauid.vhd`: inspect paging and edit-mode status word.

- `buttons(0)` = `btnD`: exit configuration and start CPU runtime monitoring

Switch usage in the design is the following:

- `sw15` (`mode_sel`): 0 = edit mode, 1 = inspect mode
- `sw14` (`page_sel`): in inspect mode selects upper/lower 16 bits on 7-seg
- `sw13..10` (`mask_in`): 4-bit byte-enable mask stored per rule (used on save)
- `sw8..7` (`sel_len`): input limit (how many nibbles `btnU` accepts during editing)
- `sw6..4` (`sel_index`): rule index (0..7)
- `sw3..0` (`nibble_in`): nibble value to append in edit mode

Visual feedback (7-seg and LEDs). The 7-seg display is driven through a 16-bit value that is selected in the top level. During configuration (`cpu_state = x"0"`) the top level displays `ui_digits` coming from `cauid`; during runtime it displays the CPU/memory digits (Figure 2). This keeps the top level clean: it only multiplexes outputs.

Inside `cauid`, the 7-seg meaning depends on the mode:

- In inspect mode (`mode_sel=1`), the selected stored rule is shown. `sw14` pages between the upper and lower 16-bit halves.
- In edit mode (`mode_sel=0`), a compact status word is shown (input counter, selected input length, rule index, and current nibble input) to reduce operator errors during manual entry.

Safety gating and runtime data capture. Configuration and inspection actions are accepted only when the CPU state is `x"0"`. During runtime states, the module ignores configuration actions so signatures cannot be modified while the CPU is executing. At

```
if (state = x"1") or (state = x"2") or (state = x"3") then
    shift_register <= shift_register(23 downto 0) & datard;

elsif state = x"4" then
```

Figure 4: State gating and runtime data capture in `cauid.vhd`: configuration only in state `x"0"`, monitoring in states `x"1"`–`x"4"`.

```
elsif state = x"4" then
    vmatch := '0';

    for i in 0 to NUM_SIG-1 loop
        m := sig_mask(i);

        if m /= "0000" then
            d := signatures(i);
            ok := '1';

            if m(0) = '1' then
                if shift_register(7 downto 0) /= d(7 downto 0) then ok := '0'; end if;
            end if;

            if m(1) = '1' then
                if shift_register(15 downto 8) /= d(15 downto 8) then ok := '0'; end if;
            end if;

            if m(2) = '1' then
                if shift_register(23 downto 16) /= d(23 downto 16) then ok := '0'; end if;
            end if;

            if m(3) = '1' then
                if shift_register(31 downto 24) /= d(31 downto 24) then ok := '0'; end if;
            end if;

            if ok = '1' then
                vmatch := '1';
                exit;
            end if;
        end if;
    end loop;
```

Figure 5: Masked comparison loop across stored signatures from `cauid.vhd`.

runtime, observed bytes are shifted into a 32-bit window in states `x"1"`, `x"2"`, and `x"3"`, and matching is evaluated in `x"4"` (Figure 4).

Masked matching and detection behavior. Each stored rule consists of a 32-bit signature and a 4-bit byte-enable mask. A mask bit enables comparison of the corresponding byte; `0000` disables the rule. When the CPU reaches the compare state (`x"4"`), `cauid` iterates through all rule entries and checks only the enabled bytes. A match sets `match=1` and stops the search at the first matching entry (Figure 5).

When a detection occurs (`match=1`), the top level forwards the signal to the CPU as `halt` and also changes the decimal point output (`dp <= not cauid_match`) as a visible indicator. If no rule matches, `match` remains 0 and the CPU continues normal execution.

Working Example

This example demonstrates how to use the configuration interface to write the signature **0xB4A9F465**, how each button behaves in practice, and how to run the CPU afterwards. The example signature is not expected to be detected unless the runtime byte window observed by CAUID matches `B4 A9 F4 65` exactly (and the corresponding bytes are enabled by the mask), as described in the Code explanation subsection.

Step 1: Select edit mode and target settings. Set `sw15=0` (edit mode). Choose a rule slot using `sw6..4`. Choose full entry length using `sw8..7=11` (8 nibbles). Enable all bytes using `sw13..10=1111`.

Step 2: Clear input once (btnL).. Press `btnL` to clear the edit buffer before entry. This ensures the next input starts from an empty buffer.

Step 3: Start entering the signature (btnU + sw3..0). Enter the signature nibble-by-nibble:

`B, 4, A, 9, F, 4, 6, 5`

For each nibble: set `sw3..0` to the value and press `btnU` once.

Step 4: Demonstrate correction using btnL.. Assume an accidental user mistake occurs (e.g., the 3rd nibble is entered as `C` instead of `A`). After noticing the incorrect input via the edit-mode feedback, press `btnL` to clear the buffer and then re-enter the full nibble sequence correctly using `btnU`. This demonstrates how the workflow handles human input errors.

Step 5: Save the rule (btnR).. Press `btnR` to save the signature into the selected rule slot together with the current mask. A save confirmation is visible via the write indicator behavior described earlier.

Step 6: Verify by inspection (inspect mode + btnU + sw14). Set `sw15=1` (inspect mode). Keep the same rule index selected (`sw6..4`). Press `btnU` to load the stored entry for display. Use `sw14=0` to view the upper half and `sw14=1` to view the lower half. For `0xB4A9F465`, the expected halves are `0xB4A9` and `0xF465`.

Step 7: Start runtime execution (btnD). Press `btnD` to exit configuration and start the CPU. In a typical run, CAUID will not detect this example because detection requires that the runtime 32-bit observation window equals `B4 A9 F4 65` at compare time (and all corresponding bytes are enabled). If the CPU program does not produce that exact 4-byte window on `datard`, then the masked comparison never succeeds and `match` remains 0.

What would happen if a detection occurred? If the CPU did produce a window matching the stored rule (subject to the mask), `match` would assert. The CPU would receive `halt=1`, and the decimal point output would change state as an alarm indicator. The system would therefore visibly indicate detection and stop the CPU according to the CPU control logic.

4.1 Assessment

The technical result of this BSP was the enhancement of the functionality of CAUID in terms of the improved customisation capability of the tool and the increased practicality of signature-based monitoring on the target platform. This is because the implemented workflow supports the interactive creation of signatures directly on the Basys3 board. This involves the provision of feedback while creating the signature and the inspection of the signature, which enables the creation of signatures of variable sizes while maintaining the structure of a comparison

The second accomplishment of the BSP development is the improved portability of the system at the integration level. This is because the configuration and inspection of CAUID are implemented within its own module whereas before it was implemented within the top layer of the CPU. This reduces the coupling of the security with a particular CPU environment, meaning that the adaptation of the module for different CPUs is mainly an issue of interfacing and routing the signals.

However, not all tasks considered for this BSP were fully completed. For instance, external signature entry via an additional interface device, such as a keyboard, was considered a task for completion, as it would have been shown more similarity to the idea of using an external hardware (realistically flash drive) to update the signature array of the chip.

Another initial goal considered for this project was to integrate CAUID with a more feature-rich CPU platform - NEORV32. This would further enhance the realism of CAUID by exposing it to more complex execution behaviour. It would allow for an evaluation of assumptions made by CAUID with respect to monitoring in an environment that more closely represents real-world processors.

Conclusion

Fully hardware-based security mechanisms are a promising direction in cybersecurity because they can move parts of monitoring outside the conventional software stack, reducing reliance on operating system integrity and making certain forms of tampering harder. However, practical deployment depends not only on detection ideas but also on scalability of signature matching and on safe, reliable configuration workflows.

In the scientific deliverable of this BSP, malware and malware signatures were clarified to motivate why signatures can be syntactic or context-dependent. The section then answered the research question **"Does supporting many signature lengths make hardware signature matching inefficient?"** by reviewing prior work on multi-pattern matching and variable-length signatures, showing that naive designs can become inefficient but that established techniques mitigate this through memory-efficient representations and prefilter-and-verify approaches.

In the technical deliverable, CAUID was enhanced to better reflect a believable signature-based monitoring workflow under realistic hardware constraints. The implementation supports interactive signature authoring and inspection on the Basys3 platform with explicit visual feedback, and it introduces variable-size signature support via a byte-enable mask mechanism. Portability was also improved by encapsulating CAUID logic within its own module and keeping the CPU top level limited to output multiplexing.

Acknowledgments

The author of this BSP expresses their gratitude to the supervisor, Bernard Steenis, for their provision of support, feedback, and suggestions for improvement during the making of this project.

References

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[3] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings of IEEE INFOCOM*, 2004.

[4] H. Song and J. W. Lockwood. Multi-pattern signature matching for hardware network intrusion detection systems. In *Proceedings of IEEE GLOBECOM*, 2005.

[5] V. Dimopoulos, G. Papadopoulos, and D. Pnevmatikatos. Split-AC: A memory-efficient version of the Aho–Corasick algorithm for reconfigurable hardware intrusion detection. *IEEE*, 2007.

[6] T. Song, W. Zhang, D. Wang, and Y. Xue. A memory efficient pattern matching architecture for network security applications. 2008.

[7] M. Souppaya and K. Scarfone. *Guide to Malware Incident Prevention and Handling for Desktops and Laptops (NIST SP 800-83 Rev. 1)*. National Institute of Standards and Technology, 2013. DOI: 10.6028/NIST.SP.800-83r1.

[8] K. Scarfone and P. Mell. *Guide to Intrusion Detection and Prevention Systems (IDPS) (NIST SP 800-94)*. National Institute of Standards and Technology, 2007.

[9] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A Semantics-Based Approach to Malware Detection. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007. DOI: 10.1145/1190216.1190270.

[10] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand. Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017. DOI: 10.14722/ndss.2017.23379.

Appendix

Project files

- `cpu_top.vhd`
- `cpu.vhd`
- `memory.vhd`
- `interface.vhd`
- `cauid.vhd`

`cpu_top.vhd`

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity cpu_top is
    Port(
        clk  : in  std_logic;
        sw   : in  std_logic_vector(15 downto 0);
        led  : out std_logic_vector(15 downto 0);
        seg  : out std_logic_vector(6 downto 0);
        dp   : out std_logic;
        an   : out std_logic_vector(3 downto 0);
        btnC : in  std_logic;
        btnU : in  std_logic;
        btnL : in  std_logic;
        btnR : in  std_logic;
        btnD : in  std_logic
    );
end cpu_top;

architecture Behavioral of cpu_top is

    component memory is
        port (
            clk, rst : in  std_logic;
          addr    : in  std_logic_vector(15 downto 0);
          datawr   : in  std_logic_vector(7 downto 0);
          datard   : out std_logic_vector(7 downto 0);
            wr       : in  std_logic;
          sw      : in  std_logic_vector(15 downto 0);
```

```
            buttons  : in  std_logic_vector(3 downto 0);
            led      : out std_logic_vector(15 downto 0);
            digits   : out std_logic_vector(15 downto 0)
        );
    end component;

    component interface is
        port (
            clk, rst : in  std_logic;
            data     : in  std_logic_vector(15 downto 0);
            segm     : out std_logic_vector(7 downto 0);
            common   : out std_logic_vector(3 downto 0);
            butin    : in  std_logic_vector(3 downto 0);
            butout   : out std_logic_vector(3 downto 0)
        );
    end component;

    component cpu is
        port (
            clk, rst    : in  std_logic;
            addr        : out std_logic_vector(15 downto 0);
            datard      : in  std_logic_vector(7 downto 0);
            datawr      : out std_logic_vector(7 downto 0);
            wr          : out std_logic;
            stateo      : out std_logic_vector(3 downto 0);
            halt        : in  std_logic;
            exit_config : in  std_logic
        );
    end component;

    component cauid is
        port (
            clk      : in  std_logic;
            rst      : in  std_logic;
            datard   : in  std_logic_vector(7 downto 0);
            state    : in  std_logic_vector(3 downto 0);
            match    : out std_logic;

            mode_sel : in  std_logic;  -- sw15
            page_sel : in  std_logic;  -- sw14
        mask_in  : in  std_logic_vector(3 downto 0); -- sw13..10
        sel_index: in  std_logic_vector(2 downto 0); -- sw6..4
        sel_len  : in  std_logic_vector(1 downto 0); -- sw8..7
        nibble_in: in  std_logic_vector(3 downto 0); -- sw3..0
            buttons  : in  std_logic_vector(3 downto 0);

            ui_digits: out std_logic_vector(15 downto 0);
            ui_leds  : out std_logic_vector(15 downto 0);
            exit_config_out : out std_logic
        );
    end component;

    signal addr           : std_logic_vector(15 downto 0);
    signal datard, datawr : std_logic_vector(7 downto 0);
    signal wr             : std_logic;

    signal led2           : std_logic_vector(15 downto 0);
    signal digits         : std_logic_vector(15 downto 0);

    signal segm           : std_logic_vector(7 downto 0);
    signal buttons_i      : std_logic_vector(3 downto 0);
    signal buttons        : std_logic_vector(3 downto 0);

    signal state          : std_logic_vector(3 downto 0);
    signal cpu_state      : std_logic_vector(3 downto 0);

    signal cauid_match     : std_logic;
    signal exit_config_sig : std_logic;

    signal ui_digits      : std_logic_vector(15 downto 0);
    signal ui_leds        : std_logic_vector(15 downto 0);

    signal digits_mux     : std_logic_vector(15 downto 0);

begin

    buttons_i <= (btnU, btnL, btnR, btnD);

    cpu_state <= state;

    digits_mux <= ui_digits when (cpu_state = x"0") else digits;
    led        <= ui_leds  when (cpu_state = x"0") else led2;
    dp         <= not cauid_match;

    seg <= segm(6 downto 0);

    cmem : memory
        port map (
            clk     => clk,
            rst     => btnC,
            addr    => addr,
            datawr  => datawr,
            datard  => datard,
            wr      => wr,
            sw      => sw,
            buttons => buttons,
            led     => led2,
            digits  => digits
        );

    cint : interface
        port map (
            clk    => clk,
            rst    => btnC,
            data   => digits_mux,
            segm   => segm,
            common => an,
            butin  => buttons_i,
            butout => buttons
        );

    ccauid : cauid
        port map (
            clk      => clk,
            rst      => btnC,
            datard   => datard,
            state    => state,
            match    => cauid_match,

            mode_sel => sw(15),
            page_sel => sw(14),
            mask_in  => sw(13 downto 10),
            sel_index=> sw(6 downto 4),
            sel_len  => sw(8 downto 7),
            nibble_in=> sw(3 downto 0),
            buttons  => buttons,

            ui_digits=> ui_digits,
            ui_leds  => ui_leds,
            exit_config_out => exit_config_sig
        );

    ccpu : cpu
        port map (
            clk         => clk,
            rst         => btnC,
            addr        => addr,
            datard      => datard,
            datawr      => datawr,
            wr          => wr,
            stateo      => state,
            halt        => cauid_match,
            exit_config => exit_config_sig
        );

end Behavioral;
```

cauid.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity cauid is
    port (
        clk      : in  std_logic;
        rst      : in  std_logic;
        datard   : in  std_logic_vector(7 downto 0);
        state    : in  std_logic_vector(3 downto 0);
        match    : out std_logic;

        mode_sel : in  std_logic;
        page_sel : in  std_logic;
        mask_in  : in  std_logic_vector(3 downto 0);
        sel_index: in  std_logic_vector(2 downto 0);
        sel_len  : in  std_logic_vector(1 downto 0);
        nibble_in: in  std_logic_vector(3 downto 0);
        buttons  : in  std_logic_vector(3 downto 0);

        ui_digits: out std_logic_vector(15 downto 0);
        ui_leds  : out std_logic_vector(15 downto 0);
        exit_config_out : out std_logic
    );
end cauid;

architecture Behavioral of cauid is

    constant NUM_SIG : integer := 8;

    type signature_array_type is array (0 to NUM_SIG-1) of std_logic_v
    type mask_array_type    is array (0 to NUM_SIG-1) of std_logic_vec

    constant DEFAULT_SIGNATURES : signature_array_type := (
        x"000000FF",
        x"0000AABB",
        x"00112233",
        x"03AA64FF",
        x"6403AC21",
        x"2A4B6C7D",
        x"94DB5312",
        x"ED1132B4"
    );

    constant DEFAULT_MASKS : mask_array_type := (
        others => "1111"
    );

    signal signatures     : signature_array_type;
    signal sig_mask       : mask_array_type;

    signal shift_register : std_logic_vector(31 downto 0);

    signal edit_data    : std_logic_vector(31 downto 0);
    signal edit_count     : unsigned(3 downto 0);

    signal display_reg   : std_logic_vector(31 downto 0);
    signal display_mask   : std_logic_vector(3 downto 0);

    signal buttons_prev   : std_logic_vector(3 downto 0);
    signal activity_tog   : std_logic;
    signal write_pulse    : std_logic;

    signal exit_cfg_reg   : std_logic;

    signal ui_digits_int  : std_logic_vector(15 downto 0);
    signal ui_leds_int   : std_logic_vector(15 downto 0);

    function need_nibbles(l : std_logic_vector(1 downto 0)) return uns
    begin
        case l is
            when "00" => return to_unsigned(2, 4);
            when "01" => return to_unsigned(4, 4);
            when "10" => return to_unsigned(6, 4);
            when others => return to_unsigned(8, 4);
        end case;
    end function;
```

```
   function len_to_digit(l : std_logic_vector(1 downto 0)) return std_logic_vector is
    begin
        case l is
            when "00" => return x"1";
            when "01" => return x"2";
            when "10" => return x"3";
            when others => return x"4";
        end case;
    end function;

   function apply_mask(d : std_logic_vector(31 downto 0); m : std_logic_vector(3 downto 0)) return std_logic_vector is
        variable r : std_logic_vector(31 downto 0);
    begin
        r := d;
      if m(0) = '0' then r(7 downto 0)   := (others => '0'); end if;
      if m(1) = '0' then r(15 downto 8)  := (others => '0'); end if;
      if m(2) = '0' then r(23 downto 16) := (others => '0'); end if;
      if m(3) = '0' then r(31 downto 24) := (others => '0'); end if;
        return r;
    end function;

begin

    exit_config_out <= exit_cfg_reg;

    ui_digits <= ui_digits_int;
    ui_leds   <= ui_leds_int;

   ui_comb : process(state, mode_sel, page_sel, display_reg, display_mask,
                edit_count, sel_len, sel_index, nibble_in,
                exit_cfg_reg, activity_tog, write_pulse)
    begin
        ui_digits_int <= (others => '0');
        ui_leds_int   <= (others => '0');

        if (state = x"0") and (mode_sel = '1') then
            if page_sel = '1' then
             ui_digits_int <= display_reg(15 downto 0);
            else
           ui_digits_int <= display_reg(31 downto 16);
            end if;

        ui_leds_int <= page_sel & exit_cfg_reg & activity_tog & "0000" & "00" & sel_index & display_mask;

        else
        ui_digits_int <= std_logic_vector(edit_count) & len_to_digit(sel_len) & ("0" & sel_index) & nibble_in;

        ui_leds_int <= write_pulse & exit_cfg_reg & activity_tog &
                  std_logic_vector(edit_count) & sel_len & sel_index & nibble_in;
        end if;
    end process;

    process(clk, rst)
        variable vmatch : std_logic;
        variable idx    : integer;
        variable need   : unsigned(3 downto 0);
```

```
        variable btnu_edge : std_logic;
        variable btnl_edge : std_logic;
        variable btnr_edge : std_logic;
        variable btnd_edge : std_logic;

        variable m : std_logic_vector(3 downto 0);
        variable d : std_logic_vector(31 downto 0);
        variable ok : std_logic;
    begin
        if rst = '1' then
            shift_register <= (others => '0');
            match          <= '0';

            signatures <= DEFAULT_SIGNATURES;
            sig_mask   <= DEFAULT_MASKS;

            edit_data    <= (others => '0');
            edit_count   <= (others => '0');
            display_reg  <= (others => '0');
            display_mask <= (others => '0');

            buttons_prev <= (others => '0');
            activity_tog <= '0';
            write_pulse  <= '0';

            exit_cfg_reg <= '0';

        elsif rising_edge(clk) then

            write_pulse <= '0';

        btnu_edge := buttons(3) and (not buttons_prev(3));
        btnl_edge := buttons(2) and (not buttons_prev(2));
        btnr_edge := buttons(1) and (not buttons_prev(1));
        btnd_edge := buttons(0) and (not buttons_prev(0));

            buttons_prev <= buttons;

        if (state = x"1") or (state = x"2") or (state = x"3") then
           shift_register <= shift_register(23 downto 0) & datard;

            elsif state = x"4" then
                vmatch := '0';

                for i in 0 to NUM_SIG-1 loop
                    m := sig_mask(i);

                    if m /= "0000" then
                        d := signatures(i);
                        ok := '1';

                        if m(0) = '1' then
                    if shift_register(7 downto 0) /= d(7 downto 0) the
                        end if;

                        if m(1) = '1' then
                    if shift_register(15 downto 8) /= d(15 downto 8) t
            end if;

            if m(2) = '1' then
        if shift_register(23 downto 16) /= d(23 downto 16) then ok := '0'; end if;
            end if;

            if m(3) = '1' then
        if shift_register(31 downto 24) /= d(31 downto 24) then ok := '0'; end if;
            end if;

            if ok = '1' then
                vmatch := '1';
                exit;
            end if;
        end if;
    end loop;

    match <= vmatch;
end if;

if state = x"0" then
   idx := to_integer(unsigned(sel_index));
    if idx < 0 then
        idx := 0;
    elsif idx >= NUM_SIG then
        idx := NUM_SIG - 1;
    end if;

    if btnd_edge = '1' then
        exit_cfg_reg <= '1';
        activity_tog <= not activity_tog;
    end if;

    if mode_sel = '0' then
        need := need_nibbles(sel_len);

        if btnl_edge = '1' then
          edit_data    <= (others => '0');
          edit_count   <= (others => '0');
         activity_tog <= not activity_tog;
        end if;

        if btnu_edge = '1' then
            if edit_count < need then
       edit_data  <= edit_data(27 downto 0) & nibble_in;
             edit_count <= edit_count + 1;
            end if;
         activity_tog <= not activity_tog;
        end if;

        if btnr_edge = '1' then
    signatures(idx) <= apply_mask(edit_data, mask_in);
            sig_mask(idx)   <= mask_in;
            write_pulse     <= '1';
       activity_tog   <= not activity_tog;
        end if;
    else
        if btnu_edge = '1' then
          display_reg  <= signatures(idx);
           display_mask <= sig_mask(idx);
         activity_tog <= not activity_tog;
        end if;
    end if;
end if;
        end if;
    end process;

end Behavioral;
```