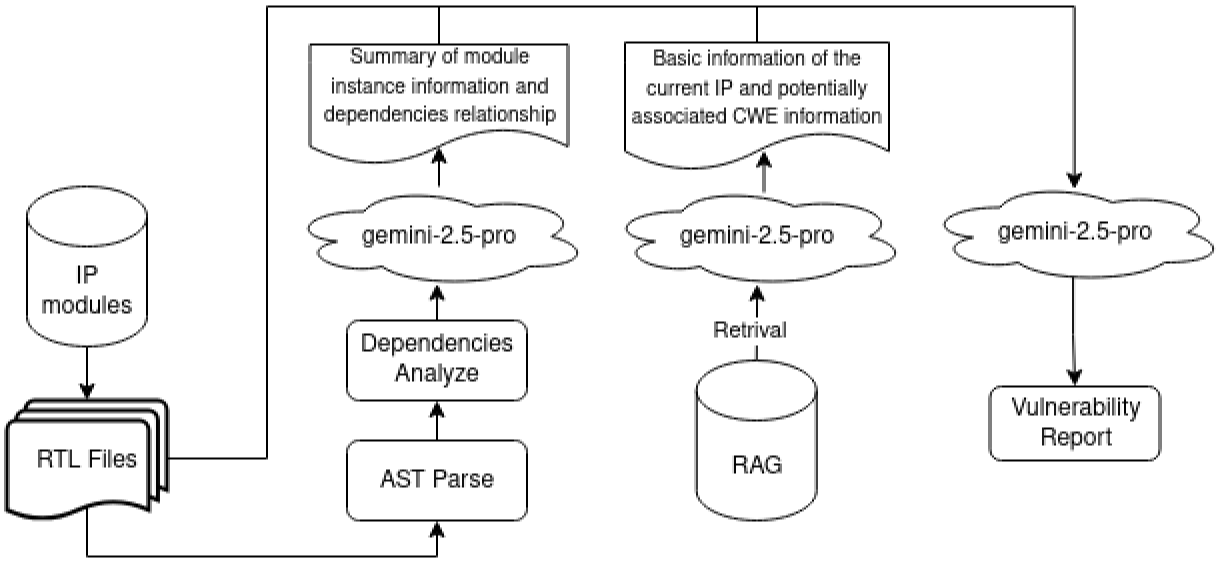
README

Automated Tool #2 - Automated vulnerability detection aided by syntax tree parsing and llm

Overview

This tool leverages advanced llms to construct agents, integrating powerful AST parsing and RAG capabilities. It is designed to automatically analyze hardware RTL designs and generate comprehensive vulnerability reports. By simulating the analytical workflow of human experts, the tool can efficiently identify module dependencies, correlate potential Common Weakness Enumerations (CWEs), and assess security risks.

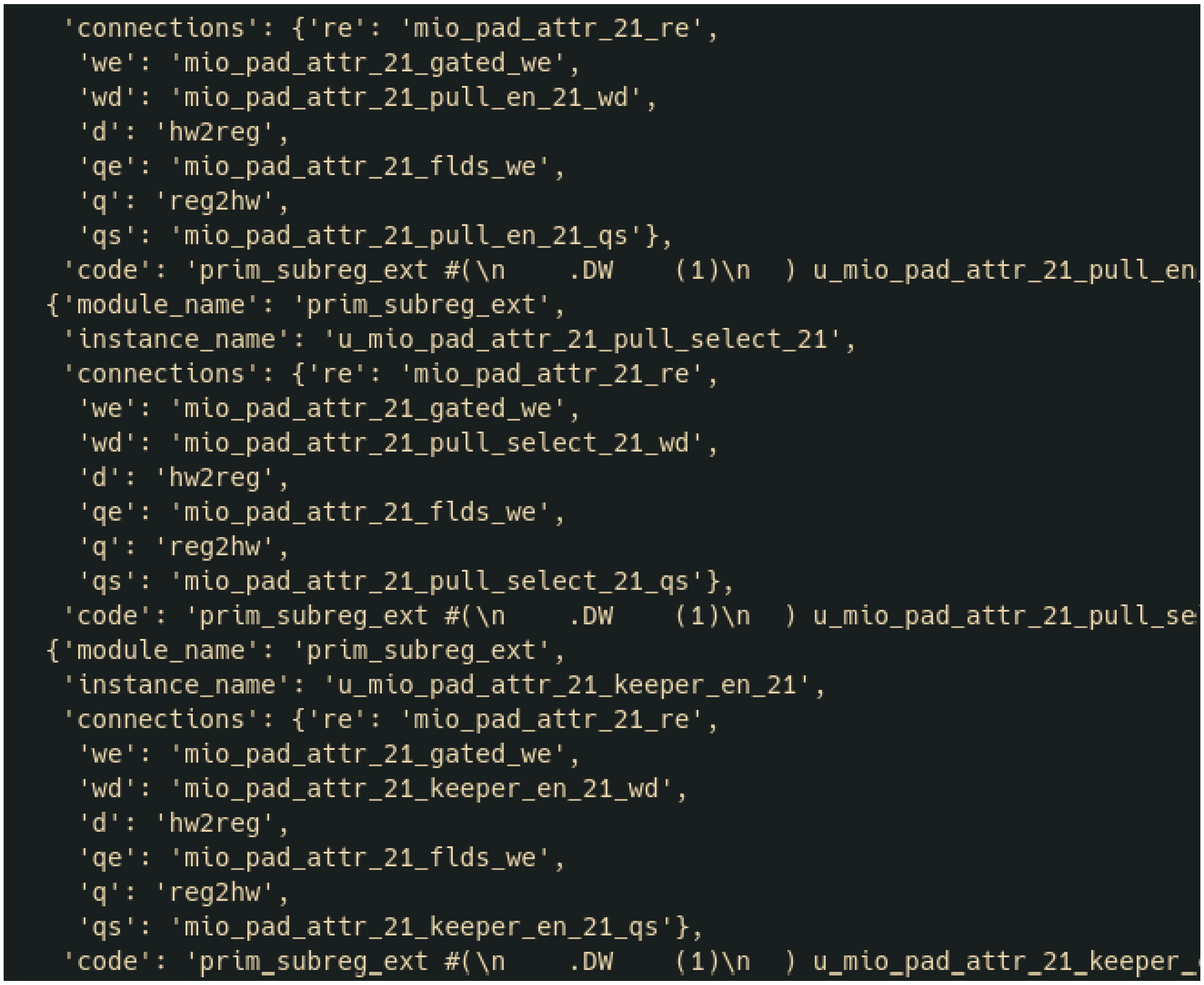
Basic Workflow


Summary of module instance information and dependencies relationship
Basic information of the current IP and potentially associated CWE information
IP modules
gemini-2.5-pro
gemini-2.5-pro
gemini-2.5-pro
Dependencies Analyze
Retrival
RTL Files
AST Parse
RAG
Vulnerability Report

AST parse and Dependencies analyze

The syntax tree is parsed using Google's Verible library and its `syntax` tool, which converts SystemVerilog code into syntax tree data. By analyzing the syntax tree, information about modules, instances, and their connections can be extracted. Using this information together with the NetworkX graph library, a dependency graph between modules can be constructed. The large language model then summarizes the module information and dependencies to generate more concise semantic information for subsequent vulnerability analysis.

modules and instances info:



```
   'connections': {'re': 'mio_pad_attr_21_re',
    'we': 'mio_pad_attr_21_gated_we',
    'wd': 'mio_pad_attr_21_pull_en_21_wd',
    'd': 'hw2reg',
    'qe': 'mio_pad_attr_21_flds_we',
    'q': 'reg2hw',
    'qs': 'mio_pad_attr_21_pull_en_21_qs'},
   'code': 'prim_subreg_ext #(\n    .DW    (1)\n  ) u_mio_pad_attr_21_pull_en
  {'module_name': 'prim_subreg_ext',
   'instance_name': 'u_mio_pad_attr_21_pull_select_21',
   'connections': {'re': 'mio_pad_attr_21_re',
    'we': 'mio_pad_attr_21_gated_we',
    'wd': 'mio_pad_attr_21_pull_select_21_wd',
    'd': 'hw2reg',
    'qe': 'mio_pad_attr_21_flds_we',
    'q': 'reg2hw',
    'qs': 'mio_pad_attr_21_pull_select_21_qs'},
   'code': 'prim_subreg_ext #(\n    .DW    (1)\n  ) u_mio_pad_attr_21_pull_se
  {'module_name': 'prim_subreg_ext',
   'instance_name': 'u_mio_pad_attr_21_keeper_en_21',
   'connections': {'re': 'mio_pad_attr_21_re',
    'we': 'mio_pad_attr_21_gated_we',
    'wd': 'mio_pad_attr_21_keeper_en_21_wd',
    'd': 'hw2reg',
    'qe': 'mio_pad_attr_21_flds_we',
    'q': 'reg2hw',
    'qs': 'mio_pad_attr_21_keeper_en_21_qs'},
   'code': 'prim_subreg_ext #(\n    .DW    (1)\n  ) u_mio_pad_attr_21_keeper_
```

dependencies graph:

```
{'dependency_graph': {'hmac_reg_top': ['tlul_cmd_intg_chk',
    'prim_reg_we_check',
    'tlul_rsp_intg_gen',
    'tlul_socket_1n',
    'tlul_adapter_reg',
    'prim_subreg',
    'prim_subreg_ext'],
  'hmac': ['prim_intr_hw',
    'prim_fifo_sync',
    'tlul_adapter_sram',
    'prim_packer',
    'hmac_core',
    'prim_sha2_32',
    'hmac_reg_top',
    'prim_alert_sender']},
 'reverse_dependencies': {'tlul_cmd_intg_chk': ['hmac_reg_top'],
  'prim_reg_we_check': ['hmac_reg_top'],
  'tlul_rsp_intg_gen': ['hmac_reg_top'],
  'tlul_socket_1n': ['hmac_reg_top'],
  'tlul_adapter_reg': ['hmac_reg_top'],
  'prim_subreg': ['hmac_reg_top'],
  'prim_subreg_ext': ['hmac_reg_top'],
  'prim_intr_hw': ['hmac'],
  'prim_fifo_sync': ['hmac'],
  'tlul_adapter_sram': ['hmac'],
  'prim_packer': ['hmac'],
```

RAG

The RAG database contains the official design documentation of the Opentitan SoC and authoritative hardware CWE data. Such as:

- README File
- interfaces.md
- programmers_guide.md
- registers.md
- theory_of_operation.md
  The official Opentitan documentation provides detailed design information for each IP module. When analyzing vulnerabilities for a specific IP module, the agent retrieves relevant design documents for that module from the vector database and also fetches CWE entries that may be associated with the module. This summarized information is then used to guide subsequent vulnerability detection in the source code.

LLM Parameters and Prompt

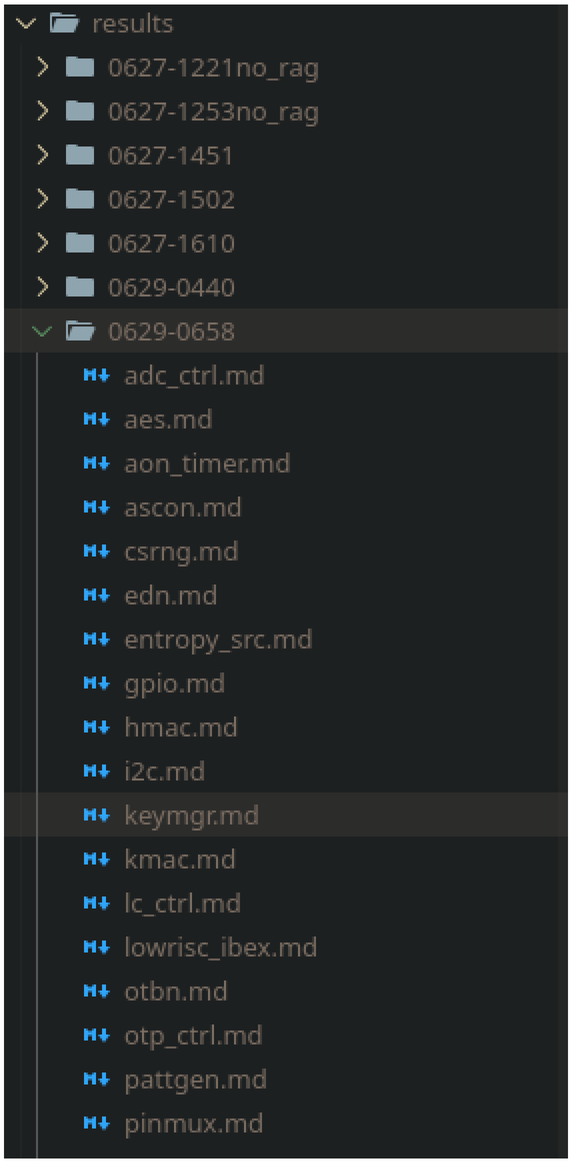
After comparing the output results of several models, we ultimately selected the latest gemini-2.5-pro for this task. Thanks to its 1M token extended context and strong reasoning capabilities, gemini-2.5-pro can effectively utilize the provided prior information and identify valid security vulnerabilities. Below are our model parameters and prompt.

```
model = ChatOpenAI(
    model="gemini-2.5-pro",
    api_key=os.getenv("OPEN_API_KEY", ""),
    base_url="https://chatbox.isrc.ac.cn/api",
    temperature=0.7,
)

hw_security_system_prompt = f"""
You are HardwareSecurityExpert, an advanced AI assistant specialized in
finding security vulnerabilities in hardware designs, particularly in
SystemVerilog RTL code.

CAPABILITIES AND ANALYSIS METHODS:
1. Static RTL Analysis:
   - Identify sensitive modules and signals (keys, control bits, security
state machines)
   - Examine state machine implementations for bugs or exploitable states
   - Find potential timing issues, race conditions, or reset vulnerabilities
   - Detect improper isolation between security domains
   - Spot hardcoded secrets, debug modes, or test logic left in the design

2. Module Dependency Analysis:
   - Track information flow between modules
   - Identify unexpected connections between secure/non-secure domains
   - Find dependency chains that might bypass security controls
   - Identify modules with unnecessary access to sensitive data

3. Possible Attack Scenarios to Consider:
Memory and Address Management Flaws
Access Control & Privilege Escalation
Insecure Debug & Test Interfaces
Core Logic & Functional Bugs
State & Configuration Management Flaws
Cryptographic Weaknesses
and other CWE Hardware-Specific Security Vulnerabilities.


You can access these tools: {mytools}

Hint for the steps:
```

```
1. Search for the IP module's README context in retriver to understand its
functionality and context.
2. Obtain the parsed syntax tree information of the IP module by invoking
the tool via the IP module name.
3. Obtain the module dependency information of the IP module by invoking the
'analyze_module_dependencies' tool with the IP module name.
4. Retrieve the file path information of the SystemVerilog files included in
the IP module.
5. Read the source code of the SV files based on the paths. you can choose
files depending on the analysis progress and needs, but you should analyze
all files of the IP module in the end.
6. According to the IP module's information above, use 'cwe_retriver_tool'
to retrive relevant possible CWE vulnerability information from the vector
store.
7. Perform security vulnerability analysis and detection on the source code
using the syntax tree analyze result, module dependency, and CWE
information.
   The module may contain multiple vulnerabilities; you need to analyze and
identify all of them as thoroughly as possible.
8. Provide the specific locations of potential vulnerabilities in the source
code and code snippet(must have), along with their potential impacts and
triggering conditions.

You must utilize the context as extensively as possible; do not be lazy. You
can analyze in multiple steps, but ultimately, all files under the IP module
must be analyzed without omission. Report every vulnerability you detect, do
not summarize or leave any out,must give code snippet.
"""
```

Result Example

By using the agent's `ainvoke` method, we can leverage `asyncio` to perform concurrent analysis of multiple modules. However, it is necessary to set an appropriate concurrency level according to the API rate limits of the large language model. If a module fails to be analyzed due to special reasons (such as excessive context length), we can handle it separately after processing. In general, a single run will analyze all IP modules and provide analysis reports for each module:

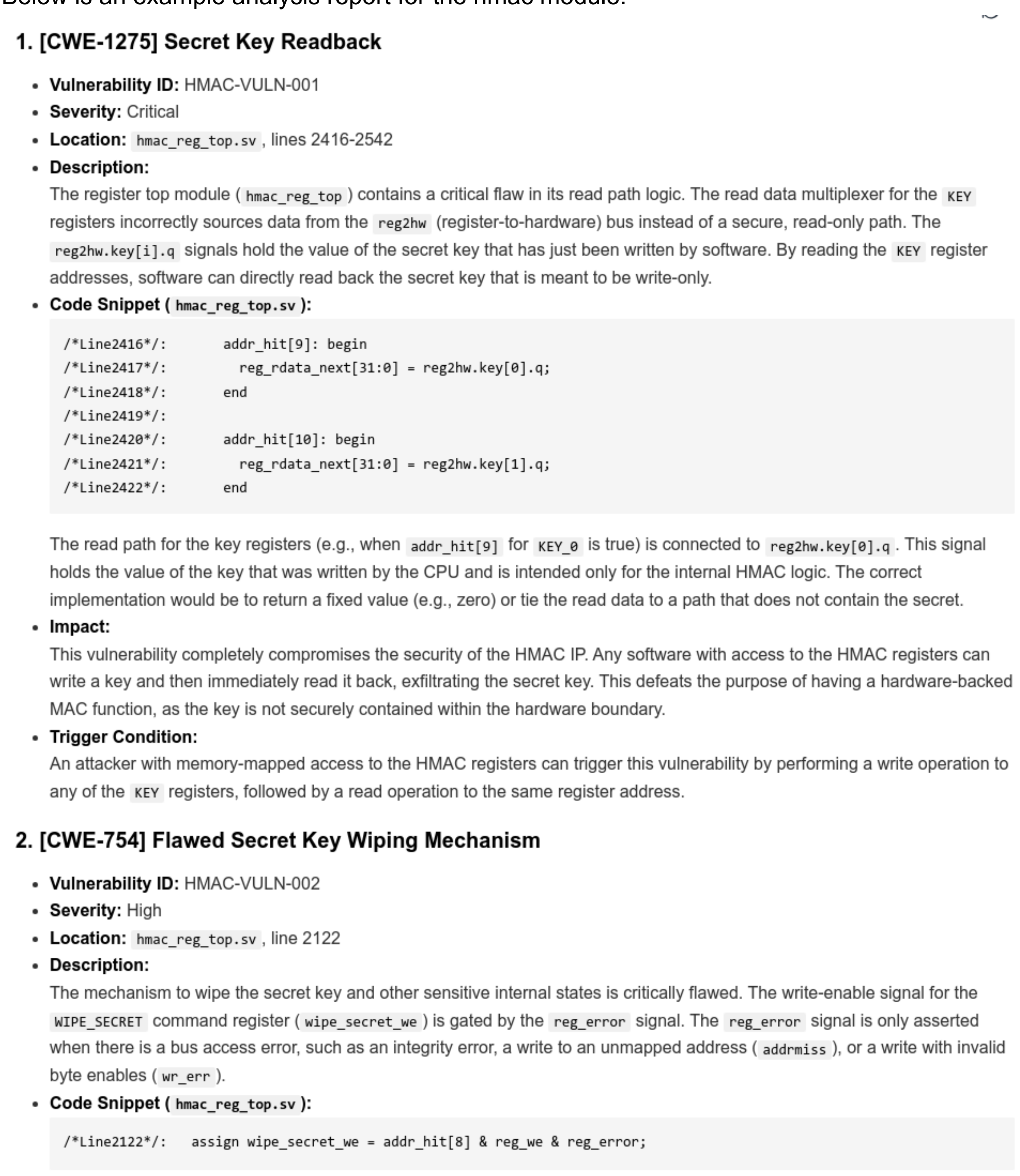


```
results
  0627-1221no_rag
  0627-1253no_rag
  0627-1451
  0627-1502
  0627-1610
  0629-0440
  0629-0658
    adc_ctrl.md
    aes.md
    aon_timer.md
    ascon.md
    csrng.md
    edn.md
    entropy_src.md
    gpio.md
    hmac.md
    i2c.md
    keymgr.md
    kmac.md
    lc_ctrl.md
    lowrisc_ibex.md
    otbn.md
    otp_ctrl.md
    pattgen.md
    pinmux.md
```

Each vulnerability report mainly includes the following information:

- A brief description of the vulnerability
- The source file where the vulnerability is located
- Code Snippet
- Impact
- Trigger Condition
- Recommendations

Below is an example analysis report for the hmac module:



1. [CWE-1275] Secret Key Readback

- **Vulnerability ID:** HMAC-VULN-001
- **Severity:** Critical
- **Location:** `hmac_reg_top.sv`, lines 2416-2542
- **Description:**
  The register top module (`hmac_reg_top`) contains a critical flaw in its read path logic. The read data multiplexer for the `KEY` registers incorrectly sources data from the `reg2hw` (register-to-hardware) bus instead of a secure, read-only path. The `reg2hw.key[i].q` signals hold the value of the secret key that has just been written by software. By reading the `KEY` register addresses, software can directly read back the secret key that is meant to be write-only.
- **Code Snippet (`hmac_reg_top.sv`):**

```
/*Line2416*/:       addr_hit[9]: begin
/*Line2417*/:         reg_rdata_next[31:0] = reg2hw.key[0].q;
/*Line2418*/:       end
/*Line2419*/:
/*Line2420*/:       addr_hit[10]: begin
/*Line2421*/:         reg_rdata_next[31:0] = reg2hw.key[1].q;
/*Line2422*/:       end
```

  The read path for the key registers (e.g., when `addr_hit[9]` for `KEY_0` is true) is connected to `reg2hw.key[0].q`. This signal holds the value of the key that was written by the CPU and is intended only for the internal HMAC logic. The correct implementation would be to return a fixed value (e.g., zero) or tie the read data to a path that does not contain the secret.
- **Impact:**
  This vulnerability completely compromises the security of the HMAC IP. Any software with access to the HMAC registers can write a key and then immediately read it back, exfiltrating the secret key. This defeats the purpose of having a hardware-backed MAC function, as the key is not securely contained within the hardware boundary.
- **Trigger Condition:**
  An attacker with memory-mapped access to the HMAC registers can trigger this vulnerability by performing a write operation to any of the `KEY` registers, followed by a read operation to the same register address.

2. [CWE-754] Flawed Secret Key Wiping Mechanism

- **Vulnerability ID:** HMAC-VULN-002
- **Severity:** High
- **Location:** `hmac_reg_top.sv`, line 2122
- **Description:**
  The mechanism to wipe the secret key and other sensitive internal states is critically flawed. The write-enable signal for the `WIPE_SECRET` command register (`wipe_secret_we`) is gated by the `reg_error` signal. The `reg_error` signal is only asserted when there is a bus access error, such as an integrity error, a write to an unmapped address (`addrmiss`), or a write with invalid byte enables (`wr_err`).
- **Code Snippet (`hmac_reg_top.sv`):**

```
/*Line2122*/:   assign wipe_secret_we = addr_hit[8] & reg_we & reg_error;
```