Undocumented Debug Interface HDT of Modern AMD CPUs

Bulat N. Zagartdinov
Institute of Cyber Intelligence Systems
National Research Nuclear University Moscow Engineering Physics Institute
Moscow, Russia
me@vaire.lt

***Abstract*—The paper discusses the implementation of the IJTAG network in the TAP controller of the modern AMD CPUs, related to the implementation of the CPU debug unlock procedure and the unlock procedure itself. During the research, reverse engineering of protocol implementation was carried out, and the structure of the hardware and software that implements the functionality of the Hardware Debug Tool (HDT) debugging interface for the AMD EPYC 7313 CPU was restored. The logic of organizing access to the TAP controller in the JTAG network based on the IEEE 1687 standard, which implements the exchange interface with the security coprocessor, was analyzed. The paper analyzed the protocol for unlocking the debugging capabilities of an undeclared hardware interface and proposed an algorithm for checking whether the debugging mode is blocked.**

Keywords—JTAG, P1687, cryptographic protocols, reverse engineering

I. Introduction

Testing is an integral stage of the development cycle. In the process of hardware development, even at the design stage, testing nodes and circuit components is laid down to reduce the production time of the final hardware solution due to timely detection of errors.

Often, developers of general-purpose hardware computing systems (CPUs and SoCs) provide interfaces for debugging software running on the computing device, which allows systems software developers to test the software being developed on the end system. In turn, AMD, as a manufacturer, has provided such an interface for its CPUs. However, no information about debug interface is currently available to the public. As a result, there is no way to verify the correct blocking of debugging capabilities in computer systems based on AMD CPUs during the operational stages, which in turn entails the risk of incorrect use of the debugging interface in production or the presence of undeclared capabilities that allow compromising the system, which must be taken into account.

Researching the implementation of a hardware debugging interface allows mitigate the risk of incorrect use in production due to the possibility of implementing a procedure for checking its status before putting the system into operation stage, as well as identifying undeclared capabilities or vulnerabilities in the implementation that allow the system to be compromised at the operational stage. This debugging interface is not currently explored due to its closed nature.

Since the most common mistake when using hardware interfaces of circuit boards in production is the complete absence or errors in the procedure for blocking them, this paper is focused on analyzing the implementation of the algorithm for unlocking the debugging interface.

II. Structure of the AMD HDT Debugging Interface

The TAP controller of the HDT interface is compatible and implements the JTAG [2], IJTAG [4] and IEEE 1500 [5] standards in various representations. JTAG is used as an interface for interacting with the TAP controller, which implements subsequent debugging logic. In turn, the TAP controller implements the internal IJTAG network through a special command that allows, by sequentially writing to a data register, to activate different sections of the circuit and vary the length of its data register. In the IJTAG network of the HDT debugging interface there are other TAP controllers whose register writing is implemented in accordance with the IEEE 1500 standard.

The HDT is divided into three parts: TAP controller in SoC with onboard interface connector, hardware adapter with RPC-based YAAP (Yet Another AMD Protocol) communication protocol, and HDT utility used to implement main CPU debug logic.

The hardware adapter that converts high-level commands from the HDT utility into low-level ones can be either an external device connected to the HDT+ onboard interface or a Baseboard Management Controller (BMC) connected to the lines of the interface through a multiplexer or resistors.

Access to the adapter device is realized via an Ethernet network with a TCP/IP protocol stack. The undocumented RPC-based YAAP protocol is an application layer protocol used to pass commands from software to the adapter device. It uses TCP as a transport protocol; by default, incoming connections are listened to on port 6123. The protocol supports authorization on the device and blocking to protect against attempts simultaneously execute commands from different user sessions.

The HDT+ onboard interface connector consists of 2 rows of 10 pins each, 5 of which are JTAG [2] interface lines - TDI, TDO, TMS, TCK and TRST. The placement of the corresponding interface contacts is shown in the Figure 1 [3].

JHDT1: 1.27mm SMD Pin header 2X10 (HDT debug)

PIN	DESCRIPTION	PIN	DESCRIPTION
1	P1V8SB	2	HDT_TCK
3	GND	4	HDT_TMS
5	GND	6	HDT_TDI
7	GND	8	HDT_TDO
9	HDT_TRST_L	10	PWROK
11	PD	12	RESET_L
13	PD	14	NC
15	PD	16	HDT_DBREQ_L
17	GND	18	NC
19	P1V8SB	20	NC

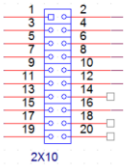

Fig. 1. Pinout of the onboard HDT+ interface connector.

The utility for CPUs of the old Kaveri architecture, which implements the functionality of the debugging interface, as

well as the user manual for it, are publicly available [1]. Corresponding software for newer CPUs is not currently publicly available. This software contains a description of the internal structure of the debugging interface for each of the supported CPUs, as well as the implementation of logical debugging operations implemented through interaction with the hardware adapter: reading registers, setting breakpoints, etc.

The TAP controller of the HDT interface is compatible and implements the JTAG [2], IJTAG [4] and IEEE 1500 [5] standards in various representations. JTAG is used as an interface for interacting with the TAP controller, which implements subsequent debugging logic. In turn, the TAP controller implements the internal IJTAG network through a special command that allows, by sequentially writing to a data register, to activate different sections of the circuit and vary the length of its data register. In the IJTAG network of the HDT debugging interface there are other TAP controllers whose register writing is implemented in accordance with the IEEE 1500 standard.

The length of the instruction register (IR) for the master TAP controller of the debug interface is 8 bits. It supports two commands: IDCODE (0x02) and SELECTDIE (0x03). In accordance with the result returned by the IDCODE command, the software selects control logic to implement the interface capabilities. The SELECTDIE command used for select an internal component for subsequent switching to its TAP controller with greater functionality, for example, I/O DIE (DIE 0) or Core Complex DIE (CCD). For AMD EPYC 7313, the IDCODE value returned is 0x0003A003. This value is related to the I/O DIE used in AMD EPYC and Ryzen Threadripper CPUs based on the Zen 3 architecture.

To access all commands in the I/O DIE TAP controller, you can select internal component (DIE) number 0 using the SELECTDIE command. The data register (DR) length for the SELECTDIE command is 6 bits. Bit 0 is the internal component selection (DIE) activation bit. Bits 1 to 4 encode the number of the component (DIE) to be selected. Bit 5 encodes the serial mode and can be activated only in conjunction with activation of all other bits.

The TAP I/O DIE controller supports many commands, for example some of them: BYPASS (0xFF), ALTTAPEN (0x05), P1687A (0x13), MANID(0x1F), AEB_STATUS (0x59), CCD_STATUS (0x63).

The CCD_STATUS command allows you to check the presence and status (enabled/disabled) of core blocks (CCD).

The AEB_STATUS command allows you to check the values of the bit fields of the platform security status register, which contains the CPU operating mode and the unlock mode state.

The MANID command allows you get more detailed information about the CPU, such as serial number, type, etc.

The BYPASS command is a standardized JTAG command and allows you to change the length of the TAP controller data register to one.

The ALTTAPEN command allows you to include additional TAP controllers in the JTAG chain. Access to this command is limited in accordance with the bits of the platform security status register.

The P1687A command is used to access the IJTAG network. For example, to access the TAP controller of the Scalable Control Fabric (SCF) into the data register (DR) of the corresponding TAP command of the I/O DIE controller of the AMD EPYC 7313 processor, it is necessary to sequentially write data 01, 01X_YYYYYYYY, where X bit selecting the next recording mode is set to 0 for subsequent writing to the data register (WDR) and 1 to the command register (WIR), YYYYYYYY is the corresponding command sent to the next TAP controller in the network. Since after a reset all select registers are activated (in state 1), first write will be made to the command register. In this case the length of the command register (IR) of the SCF controller is 8 bits. The first 2 bits deactivate the SIB Exclusion Bit (SEB) register and activate the SIB Exclusion Bit (SIB) register. In this case, the chain contains only one segment with the TAP controller SCF, then access to subsequent network segments is provided through it. The SEB register is used to exclude a group of SIB registers from the chain, thereby reducing the length of the data register.

The software uses access to the MP0_J2P_MBX0/ MP0_J2P_MBX1 interface registers of security coprocessor to implement the unlocking procedure. By writing or reading these registers, communication protocol with the security coprocessor through the HDT debug interface implemented.

Figure 2 shows a diagram of access to interface registers in the IJTAG network for the AMD EPYC 7313 CPU. The root node in the diagram is the component labeled “design_iod”, which is the TAP I/O DIE controller. Access to further network elements provided through the command data register P1687A of this controller.

In order to simplify the SEB access scheme, registers, write mode selection registers, command and data registers are not displayed on it.

In the BC1500_NETWORK network segment, nodes "smu_mp0_t" and "smu_ccp_t" are controlled using a general write mode selection register; writing to the command or data register is carried out for two devices simultaneously.

Components R01-R09 act as routers in the IJTAG network and are used to split the network into smaller segments.

III. Debug Interface Unlock Protocol

Components R01-R09 act as routers in the IJTAG network and are used to split the network into smaller segments.

The utility that implements the unlocking procedure directly in the process is not involved, but is a node that ensures interactions between the server and the security coprocessor software. It is omitted from the description of the unlocking protocol presented below.

The debug interface unlocking protocol is implemented by the finite state machine in the security coprocessor software, as well as by the server as follows:

1. The security coprocessor software generates a 32-bit random number and signs it with a private key generated based on the hardware configuration with module length 2048.

The correctness of the generator state is not checked in the procedure for generating a random number; if the hardware random number generator fails or there is insufficient entropy

to generate a random number (for example, as a result of external influence), the signed number will be zero.

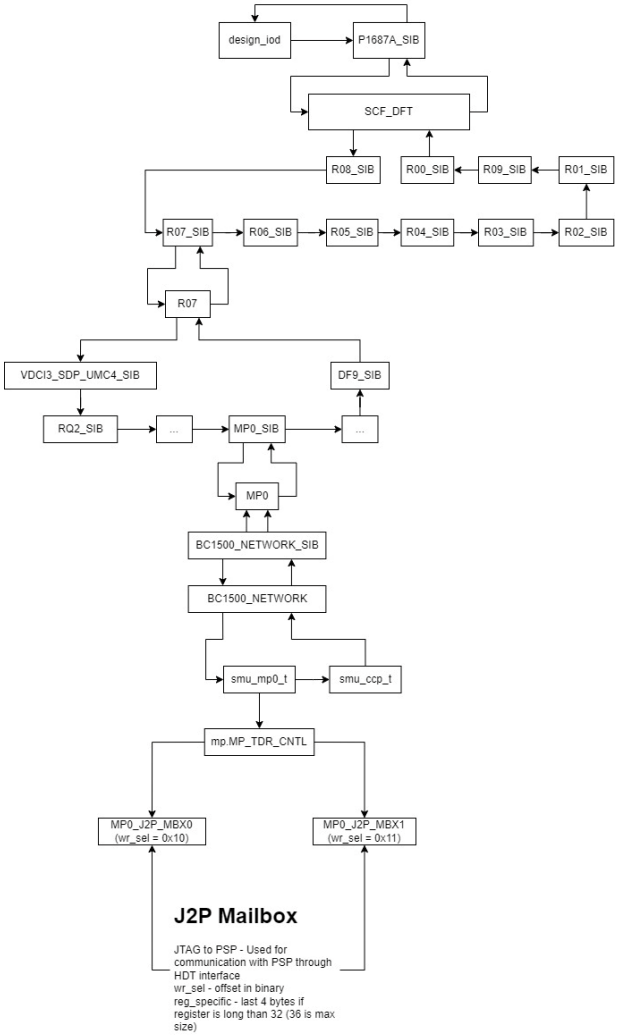

Fig. 2. Diagram of the access path to the interface registers.

2. The signed random number with the signature is sent to the server, which verifies the received value.

Communication between the security coprocessor and the software implemented by setting bit flags in the MP0_J2P_MBX0 register and then writing (reading) a piece of data to the MP0_J2P_MBX1 register.

The size of the data transmitted at this step is 264 bytes: 8 bytes of a random number and 256 bytes of its signature.

3. The server signs the unlock token on a special debug token signing key according to the RSA PSS scheme and sends it to the security coprocessor software.

The size of the unlock token is 26 bytes, the size of the signature is 512 bytes. The debug token contains the platform serial number, unlock type, AEB register and peripheral access parameters register value.

The structure of the debug token is presented in Table I.

4. The coprocessor software checks the received token against the debug token signing key. This key is stored in external flash memory and signed with the root key of the platform.

TABLE I. DEBUG UNLOCK TOKEN STRUCTURE

Offset	Size	Description
0x6	0x2	Protocol version
0x8	0x4	Peripheral Debug Modules Access Register
0xC	0x4	Peripheral Debug Modules Access Register
0x10	0x1	JTAG debug control register bits
0x11	0x1	Unlock type (not used)
0x12	0x8	Serial number (not used)

Before checking the token signature, the software verifies the protocol version used to communicate with the server, located in bytes 6 and 7 of the received token, for equality to the value 0x212. The first 8 bytes of the received token is not involved in the signature verification procedure because if verification successful, software overwrites them with a previously generated random number. Also, before checking signature, the size of the received token is checked to ensure it is equal to the value 0x21A.

5. If verification is successful, the token is synchronized with all child cores of security coprocessor. The encryption keys stored in the cryptographic coprocessor are cleared, and the debugging capabilities provided in the token are unlocked.

The platform supports prohibiting the ability to unlock debug mode, but this feature is implemented in software.

Checking the serial number field in the security coprocessor software during the unlocking procedure is also not implemented, which, together with the lack of checking the state of the hardware random number generator, makes it possible to implement an attack on using a token signed by the server on other platforms with the same debug token signing key. Due to the absence in the token structure of any fields related to the lifetime of the token, obtaining a single token signed by the server with a zero random number allows compromising all platforms with the same certificate (the entire family of processors) in the event of an attack on the hardware random number generator.

The constants used in step 1 to initialize the pseudorandom number generator (for certificate generation) can be obtained from unencrypted platform software, allowing a token acquisition attack for a null number signing without using any platform. Nonetheless, to implement such an attack, you must have a valid account on the AMD server used to generate tokens with the ability to unlock at least one platform.

If code execution can be achieved in the security coprocessor, debug mode can be unlocked by writing to several hardware registers without an unlocking procedure. It can be done, for example, by downgrading the security coprocessor software version to the vulnerable version and then exploiting a known vulnerability to execute arbitrary code in the security coprocessor. Rolling back the version is possible because the internal bootloader does not have a procedure for checking the version of the external bootloader being loaded.

In the same time, unlocking debugging capabilities can be achieved with hardware access through the fault injection attack discussed in [6].

IV. Debug Interface State Checking

The algorithm for checking the state of the debug interface in the general case can be implemented by checking the state of the AEB register. It can be accessed by sequentially executing the JTAG commands SELECTDIE and AEB_STATUS. Implementation of the algorithm for checking the state of the debugging interface in SVF syntax is proposed on the Figure 3.

```
SIR 8 TDI (03);
SDR 6 TDI (1);
SIR 8 TDI (59);
SDR 43 TDI (0000000000) TDO (0000000000) MASK(0000000030);
```

Fig. 3. Algorithm for checking debugging interface locking.

Bits 4 and 5 of the AEB register are responsible for unlocking the debugging capabilities of the platform; if any debugging capabilities are activated, these bits will be activated (equal to 1), which will be detected by the algorithm.

V. Conclusion

The logic for organizing access to the TAP controller, which implements the exchange interface with the security coprocessor for a unlocking debugging capabilities protocol, has been restored. Access to the controller that implements the communication interface with the security coprocessor is realized by sequential activation of the corresponding zones of the JTAG network by changing the SIB registers.

In the unlocking protocol implementation in the security coprocessor software several mistakes were discovered, which allows unlocking an arbitrary platform with a signed zero random number token. Since the implementation of such an attack involves influencing the hardware random number generator of the platform in order to temporarily disable it or exhaust its internal source of entropy as a result of a large number of requests, without the presence of a primitive that allows influencing the hardware random number generator, exploitation of this flaw is virtually impossible. Future studies may explore ways to manipulate the hardware random number generator.

The lack of verification by the internal loader of the version of the security coprocessor software loaded from external memory allows a rollback attack to the software version to a vulnerable one and, through its exploitation, activating the debugging mode. Such an attack can be done remotely, since the BMC has write access to external flash memory and the debugging interface.

An implementation of an algorithm for checking the state of the debugging interface in SVF syntax is proposed. It can help mitigate the risk of incorrect use of the debugging interface in production.

References

[1] KaveriPI AMD BIOSDBG tool, [online] Available: https://github.com/fishbaoz/KaveriPI/tree/master/Tools/AMD%20BIOSDBG.

[2] "IEEE 1149.1", [online] Available: http://grouper.ieee.org/groups/1149/1/.

[3] Lanner NCA-4112 User Manual, [online] Available: https://www.lannerinc.com/support/download-center/user-manuals/category/16-network-appliances?download=489:nca-4112-user-manual.

[4] "IEEE P1687", [online] Available: http://grouper.ieee.org/groups/1687/.

[5] "IEEE 1500", [online] Available: http://grouper.ieee.org/groups/1500/.

[6] R. Buhren, H.-N. Jacob, T. Krachenfels and J.-P. Seifert, "One glitch to rule them all: Fault injection attacks against AMD's secure encrypted virtualization", Proc. ACM SIGSAC Conf. Comput. Commun. Secur., pp. 2875-2889, Nov. 2021.