


RESEARCH

Open Access



Adaptive security monitoring for next-generation routers

Christopher Mansour and Danai Chasaki* 

Abstract

In today's Internet, modern routers rely on high-performance reliable general-purpose multi-core packet processing systems in order to support the flexibility and the plethora of protocol operations and applications. These processing systems are programmable and have replaced the traditional-fixed logic ASICs in the data path of such routers. Hence, lots of vulnerabilities and faults are introduced as the result of such programmability making the systems susceptible to attacks and failures. Particularly, it is a difficult task to detect whether a processing core behaves correctly, or it has a failure resulting from errors or attacks. In this paper, we address this problem by proposing a novel approach to verify the correct operation of the network processor. We propose a secure, fault-tolerant, and reliable monitoring subsystem which functions in parallel with the processing core of the router and aids in the detection of attacks changing the processing behavior of the processor. We prove experimentally that our system has the ability to detect the malicious activity and securely restore the router's operation to a different, but functionally equivalent, state. We also show experimentally that our approach has a better efficiency when compared with other existing work.

Keywords: Communications, Monitors, Network security

1 Introduction

The size, diversity, and complexity of modern networks continue to increase resulting in the development of new protocol and communication paradigms, such as content-addressable networks, that need to be deployed in order to improve the network operation. This entails the use of programmable network processors to meet such demands. Fortunately, the advances in the performance of general-purpose multi-core processors fulfilled this necessity by enabling the development of routers which are based on highly parallel, embedded multiprocessor systems-on-chip (MPSoCs) as an integral component. Network vendors and operators can now achieve a new level of flexibility due to the use of programmable components in the data path [1]. In the past, to achieve the performance and speed necessary for traffic forwarding, high-performance routers used application-specific integrated circuits (ASICs) to implement forwarding systems. Even though this technology was costly to develop, it represented the only way to achieve the required speed

and performance. Such technology did not present any potential target for attacks because once they are designed, their functionality cannot be changed except by replacing them with new hardware. However, by introducing programmability to the data plane of such routers, this premise has changed. The general-purpose processors do exhibit the same kind of vulnerabilities that have been observed and exploited in conventional end systems and embedded systems making them target for attacks.

In today's Internet, software is becoming an integral part with the emergence of software-defined networking (SDN) [2] and network function virtualization [3] introducing a lot of vulnerabilities. These vulnerabilities are due to the weaknesses that exist in the trusted code that might already be present in the system. Vulnerabilities in the network infrastructure are particularly problematic. This is because routers are shared infrastructure, and outages can affect a large number of users. Additionally, attacks targeting such vulnerabilities can undermine special-permission protocols and thus gain access to sensitive data or system resources. The existing intrusion detection and prevention mechanisms largely target

* Correspondence: danai.chasaki@villanova.edu

Department of Electrical and Computer Engineering, Villanova University, 800 Lancaster Ave, Villanova, PA 19085, USA

end systems [4]. Thus, they are insufficient against novel attacks that target the vulnerabilities in such network devices. Attackers may target the data plane of modern routers to interrupt the service and perform malicious activities. A carefully crafted packet from an attacker might exploit a software vulnerability and cause a complete system crash [5]. Considering the potential impact of a denial-of-service attack launched from a core router connected to dozens of links with 40 Gbps data rates, it becomes clear that there is a need to protect these systems.

A wide range of hardware monitoring techniques have been proposed in order to reduce the vulnerabilities existing in embedded systems [6] and packet processing systems [7]. Most of such approaches use the information about the correct processor execution tracking the instructions being executed by the processing core. An attack on the processor will then be detected by using such information, and the processor will be restored to its initial state. However, the recovery mechanism in such approaches does not eliminate the vulnerability that caused the attack in the first place, even if the processor has been restored to operate correctly. Thus, an attacker could keep sending attack packets, initiating the recovery process more than once and causing unnecessary overhead to the processor's operation.

In this paper, we propose a novel monitoring technique that monitors the network processor's operation and checks whether the processor's execution trace falls within the allowed behavior or not. We base our monitoring design on the concept of software diversity to enhance the security requirement and to ensure a high level of adaptivity and resilience. The software diversity in our design will help in generating several versions of the program to be executed. This offers different versions of the code that are functionally equivalent and can be used after the recovery process. Hence, this will eliminate the possibility of targeting the same vulnerability and preventing the processor from completing the required functionality.

The specific contributions of this paper are as follows:

- We present a new hardware monitoring system that can co-exist with the networking processor, check its execution trace, and detect unusual activities based on the instruction execution flow.
- We implement the monitor using ternary content-addressable memory (TCAM) to ensure fast detection and recovery mechanisms.
- We build our design on the concept of software diversity in order to ensure its adaptivity, reliability, and resilience.
- We implement an integrity checking mechanism using SHA-256 [8] hash to check the code integrity before execution, thus eliminating the possibility of instruction memory modifications and bit flips.
- We present experimental comparison between our design and existing techniques to show how our design is able to detect attacks and malicious activity that changes the processor's execution flow more efficiently with less resource requirements.
- We present experimental results from a prototype implementation of our monitor on a NetFPGA to demonstrate its activity and the way it can detect an attack and initiate a recovery process.

Overall, we believe that this system provides a novel efficient approach in the detection of malicious behavior of network processors in general. Additionally, it does not only detect an unusual activity but rather can initiate a secure fast recovery process.

The remainder of the paper is organized as follows: Section 2 discusses some literature review and presents experimental comparisons with other existing techniques. We discuss the security model we implement in Section 3. In Section 4, we present our adaptive monitor design. Section 5 describes the evaluation process. Resource utilization is presented in Section 5.3. Section 6 summarizes and concludes this paper.

2 Literature review

2.1 Related work

Most of the security issues in networking are related to end systems and protocols. Research in network security has focused on many topics ranging from secure end-to-end protocols, such as IPsec [9] to anomaly detection [10]. Packet marking strategies have been also proposed in order to identify attack sources [11] and protect against denial-of-service attacks. From the network side, firewalls [4] and intrusion detection systems [12] can protect some systems from some known attacks.

On end systems, virus scanner software can also identify some other attacks. However, using a virus scanner software as a defense mechanism against intrusion assumes that a sufficiently powerful processor and operating system are available. This assumption does not hold when considering embedded packet processors on routers. These systems frequently use network processors, which are embedded multi-core systems-on-a-chip that operate without operating system support to maximize throughput performance. These embedded processing systems are vulnerable to intrusion just as conventional end systems are [13].

When addressing security issues in the network infrastructure itself, very little work can be found in the literature. The study in [13] surveyed network devices that are vulnerable due to exposed interfaces which are part

of the control plane and can be protected by better management methods. However, in our work, we consider the data plane which inherently needs to be exposed and thus propose a novel protection technique. Some defenses may be based on techniques from embedded system security [14]. Other defenses are based on monitoring.

Several processor monitoring techniques have been proposed in the literature. In [15], Tokuda et al. proposed a monitoring technique by installing a special software on each processing core; this software can be a code instrumentation inside the network processing application or a dedicated code executed outside of the application. The monitoring information is then communicated to a central control processor using the same interconnect as that of the processing core for moving packets and other data. Such monitors require processing resources on the network processor and thus reduce the overall system performance. Additionally, software monitors require modification to the application binary and other additional specialized codes and does not scale well. Furthermore, monitoring techniques based on software are themselves pieces of code that can be targeted by attacks and thus are vulnerable to corruption [6]. However, it is important to note that our hardware monitor does not execute any code itself; rather, it only ensures that the processor is behaving correctly and going through the expected legitimate instructions. With the holistic view that the SDN offers, several solutions were proposed to monitor and detect network attacks by collecting the network statistics. In [16], the authors propose a flow-graph model learned from SDN messages to detect network level attacks on the network topology and the data plane forwarding. In [17], Braga et al. suggest an application to monitor the network flows to detect network flooding attacks. Similarly, netfuse [18] was proposed in order to monitor the network and find suspicious flows. Unlike such approaches, which act at the application layer of the SDN hierarchy, our hardware monitor acts at the data plane layer to detect attacks and prevent recursive in-network attacks that target the same vulnerability. Additionally, the aforementioned monitoring approaches do not detect attacks that change the instruction execution of the network processor [7] since they only rely on the flow statistics; however, our monitoring approach ensures that the right execution flow is being followed by the network processor. Furthermore, the monitoring approaches mentioned will be software applications running on top of an operating system (Network Operating System such as NOX [19], etc.). Hence, they need the services and functions provided by the OS to operate and thus will have less performance compared to our monitoring approach which acts at the data plane layer without

any service requirements from the network operating system.

In terms of hardware monitoring techniques, Mao and Wolf [20] proposed a hardware monitor for embedded systems that can track each instruction of the processor and compare it to the processing model used by the monitor. In [21], Ragel et al. presented a novel hardware/software technique at the granularity of micro-instructions to reduce overheads considerably. Arora et al. [6] presented a hardware-assisted paradigm to enhance embedded system security by detecting and preventing unintended program behavior. Similarly, the work proposed in [22] determines correct operation based on a block of instructions. Unlike our monitor, such approaches operate at the granularity of basic blocks, thus requiring more memory resources for the monitoring system, are slower in detecting attacks, and require more resources in terms of memory and execution time. A detailed comparison between our hardware monitoring design and existing techniques is provided in the following section.

Other techniques [23, 24] extend the processor instruction set and micro-architecture to support special verification steps. In [25], Chen et al. proposed an approach to verify the correct operation of packet processors by analyzing the packet latency and throughput. In [26], Mansour and Chasaki proposed an approach to detect faults and attacks in network processors through power monitoring. Our work extends the idea of hardware monitoring further and enhances it. The monitor we propose is adaptive, fault-tolerant, and reliable because it utilizes the idea of software diversity. It is also secure because the code integrity is checked before execution. Attacks in our design can be detected within a few instructions rather than at the end of a longer code block. The detection and recovery are fast processes because the monitor is based on a TCAM memory.

This article is an extension of the work that has been presented previously in a short poster paper [27] and in [28].

3 Framework security model

A best practice when designing a security design is to have a security model. Therefore, the monitoring system we present implements a security model which reflects the operation of the current Internet. We assume that the initial code on the router is benign and an attacker aims to modify the code maliciously to perform malicious activities. Such malicious activities may include a stack smashing/buffer overflow attack that can corrupt local variables and the function's return address and thus might return to another malicious code hidden inside the packet payload [5, 7].

For a secure packet processing system, there exist the following security requirements [7]:

- The network processor should not deviate from any normal forwarding behavior. The network processor should always execute the instructions that are loaded to the instruction memory. No other instructions should be executed.
- Any malicious attempts through the data plane should be detected and lead to a packet drop. Malicious attempts should be detected by our monitor, and a recovery mechanism should be initiated to restore the processor to a secure functional state.
- If an intrusion was successful and was able to change the internal state of the processor, a recovery mechanism should reset the router into an equivalent functional state. This can be done by first dropping the packet that started the malicious activity by causing the network processor to deviate from the expected behavior.
- Recovery overhead from malicious attempts should not lead to a denial-of-service.

In the context of our security model, we assume that an attacker is able to perform the following actions:

- Send abnormal packets to the network processor to trigger a malicious behavior. An example of such a behavior would be a buffer overflow on the packet processor. This latter can lead to a stack-smashing attack, which can be used to modify the control flow of the packet processing program redirecting the control flow to a malicious piece of code contained in the packet payload.
- Gain remote access to the system in which he can change the memory contents of the instruction memory, log files, or extract and modify secret keys.
- Launch a denial-of-service by resending the malicious packets over and over. Such malicious packets will always be detected by the monitor we are proposing.
- Use reprogramming interfaces to control the router.

It is important to note that attackers do not have physical access to the router and thus cannot access the binary files of the application being loaded and executed on the packet processor. This is because such files reside outside the platform. However, once the application binaries are loaded on the instruction memory, the latter is considered to be a potential attack target.

One potential attack example could be as follows: An adversary transmits a data packet that contains malicious code (e.g., within the packet header or the packet

payload). If the packet is carefully crafted, a buffer overflow on the network processor may occur. This leads to a stack-smashing attack, which can be used to modify the control flow of the packet processing program. A possible target for the redirected control flow is the malicious piece of code contained in the packet. If that code is executed, the attacker can execute arbitrary operations on the packet processor causing denial-of-service. The crafted packet will target a particular version of the code that can be avoided by another version through diversity.

In our design, the existence of the hardware monitor detects the abnormal packets and initiates a fast and secure recovery process. The code integrity check we perform before code execution eliminates the scenario of instruction memory modification. Furthermore, the integration of software diversity into our design eliminates the denial-of-service scenario since a new version of the code would have been loaded and will not be affected by the same vulnerability.

Furthermore, since our monitoring components are embedded in the system hardware, it is difficult for an attacker to attack both the processor and the monitor (which is hard to access) at the same time. Thus, this approach by design provides more security than a conventional general-purpose processing system.

4 Monitor design

In this section, we present the adaptive design of our monitoring subsystem which validates the router's correct operation by monitoring the "processing flow" of the network processor. The overall system architecture is presented in Fig. 1, and the logic operation is presented in Fig. 2. Our design constitutes of two phases, an offline phase and a runtime phase. In the offline phase, we apply the concept of software diversity and prepare the monitor. The runtime phase includes the monitoring operation in parallel with the processing flow and the recovery operation which is triggered when a malicious behavior is detected. Additionally, at the beginning of the runtime phase and after each recovery, a code integrity check occurs to verify the integrity of the code being loaded to the instruction memory and eliminate the possibility of memory modification.

When designing the hardware monitor, several challenges were taken into consideration:

- Correct detection: it is an important specification for a processor monitor to be effective. The design we propose achieves this by checking for any deviation from the expected operations prepared during the offline analysis.
- Fast detection: intrusions should be detected quickly in order to reduce or even eliminate their impact on

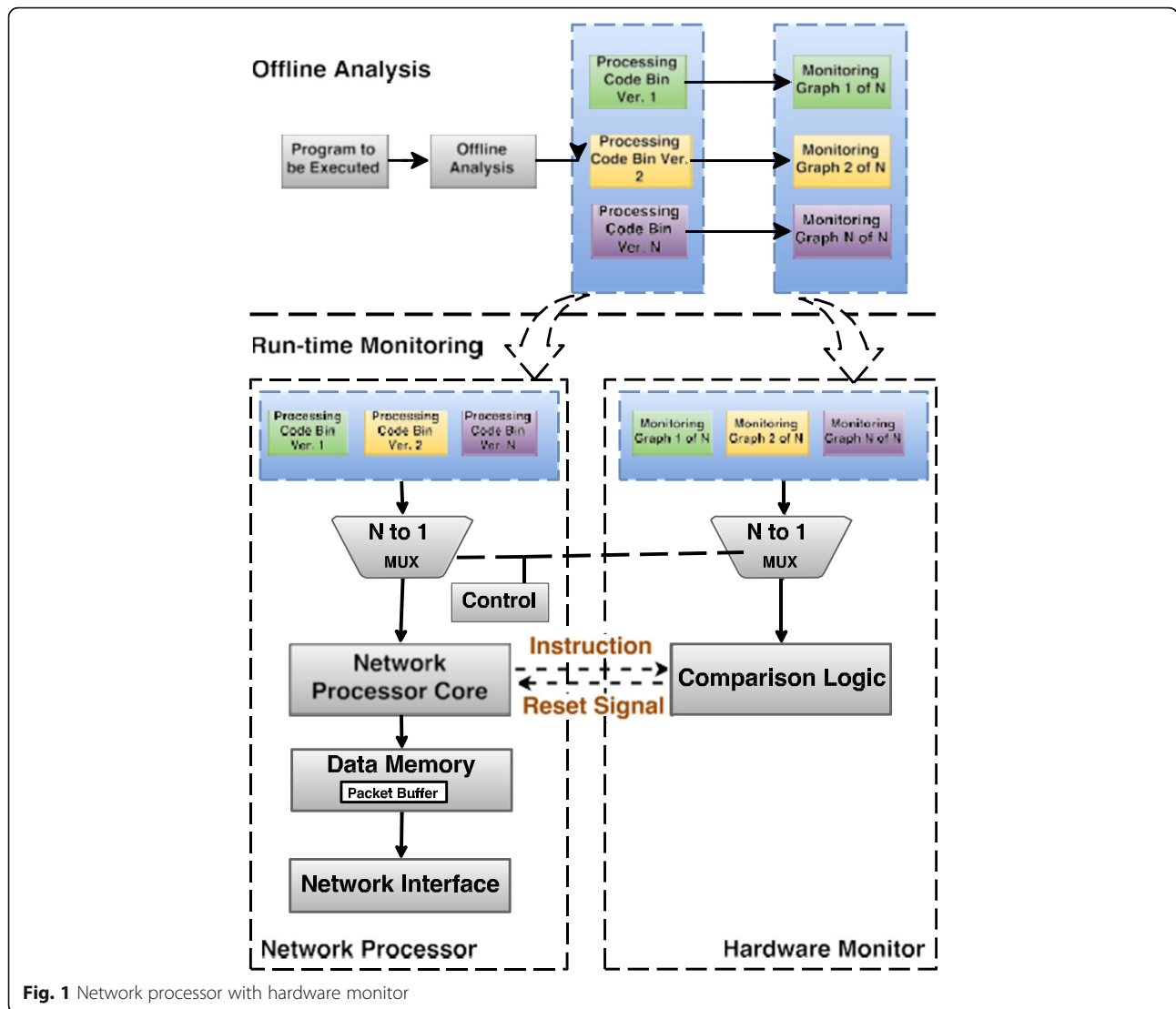


Fig. 1 Network processor with hardware monitor

the network performance. The design we propose is fast because it uses the TCAM memory.

- Low overhead: resources in a network processor is an important factor in order to limit the hardware implementation cost. The design we propose unlike other existing techniques requires a small percentage of resources.

4.1 Software diversity in network processor

Software design diversity is not a new concept; there are several ways to create different multiple software alternatives that are functionally the same [29]. An example of software diversity in C language is provided in Figs. 3 and 4. Figure 3 represents a sample code to copy a string from a source buffer “src” to a destination buffer “dst.” The same functionality happens in the sample code of Fig. 4. However, the latter is more secure because it performs dynamic memory allocation unlike the former

which does not check for the size of the source buffer. Hence, if an attacker targeted the vulnerability existing in the first version of the code, the second version will not be affected.

In the context of data plane attacks in routers, which exploit protocol processing vulnerabilities, the benefit of installing N versions of the protocol code is two-fold:

- The attacker has to spend a considerable effort to craft different attack packets to target the particular code vulnerabilities in multiple software designs.
- If only one version of the protocol binary is present, the same protocol code will be reloaded during recovery. However, with the use of software diversity, N versions of the code exist, and thus, the system can switch to a different version to avoid subsequent attempts to stall the system by sending the same attack packet over and over.

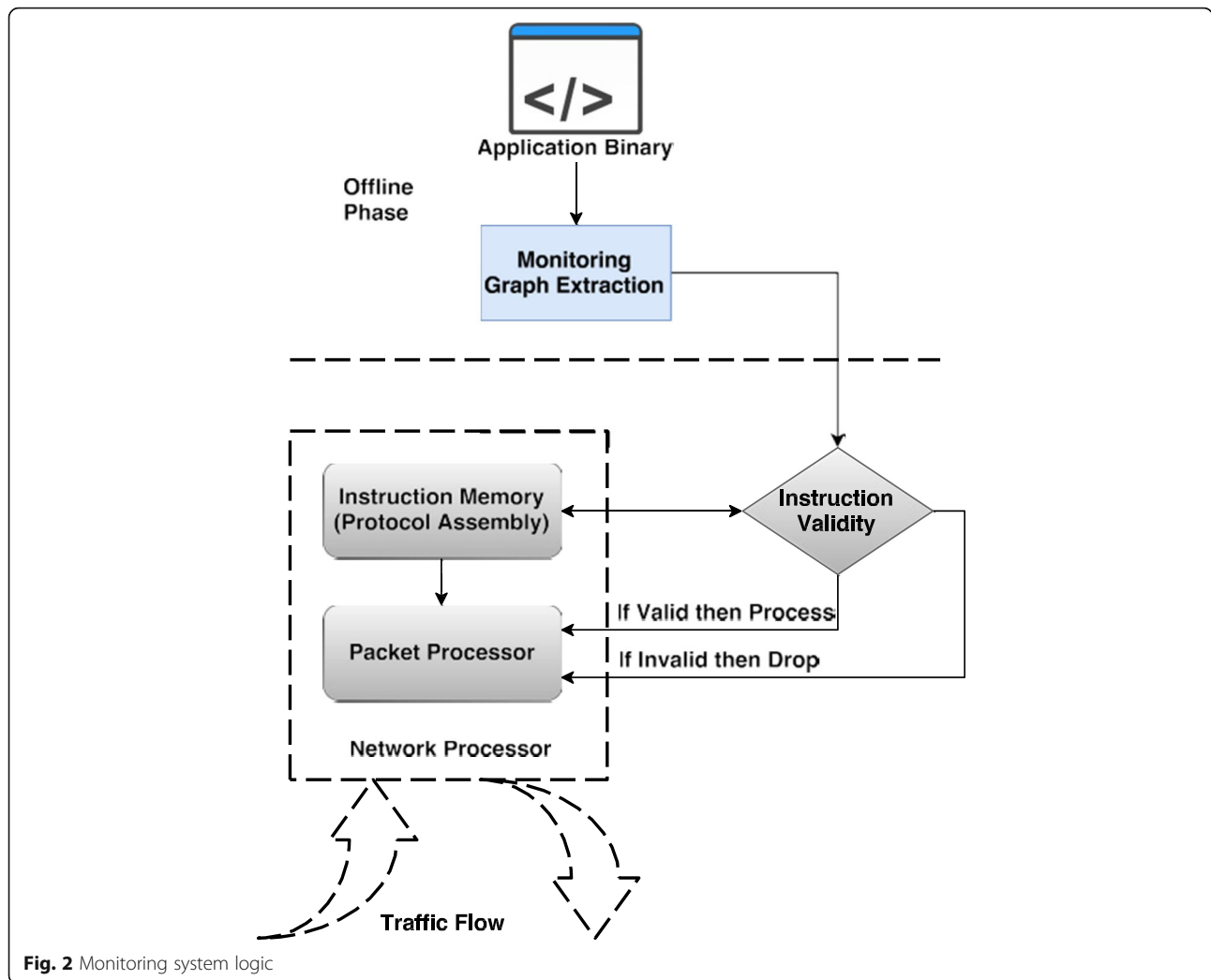


Fig. 2 Monitoring system logic

Figures 5 and 6 show the implementation of software diversity. They show two different instruction sequences for the same protocol, the IP-Forwarding in this example. Having different instruction sequence leads to a different monitoring graph. Furthermore, different instruction sequence means different execution flow and thus prevents recursive in-network attacks triggered by sending the same attack packet over and over. As an example, if a packet targeted the sequence of instructions

presented in Fig. 5, it would not be able to affect the sequence of instructions presented in Fig. 6; thus, the router functionality did not change (since both instruction sequences correspond to the same function) but the vulnerability was eliminated.

The importance of software diversity in our design is that if one version of the code was a target for an attack through a carefully crafted packet, the packet will be discarded, the processing stack will be reset, and a new version of the code will be loaded. The same attack packet will not be able to affect the new version of the code, since carefully crafted packets take advantage of specific code vulnerabilities that will most likely not be present in newer version of the code (e.g., buffer size allocation through array instantiation vs. using malloc). This makes such attacks harder since the attacker has to craft different packets to target the different versions of the code making the attack infeasible.

In our implementation, we modify the original C code of the protocol under test and then compiled it.

```

void function (char * src)
{
    char dst[32];
    strcpy(dst, src);
}
    
```

Fig. 3 Sample software diversity in C: copying a string version 1

```

void function (char * src)
{
    char * dst;
    int len = strlen(src) + 1;
    dst = (char *) malloc(len * sizeof(char));
    strcpy(dst, src);
}

```

Fig. 4 Sample software diversity in C: copying a string version 2

Checking the assembly code reveals a change in the opcodes or immediate values or even both in some cases.

4.2 Monitoring graph generation

Figure 1 shows a network processor protected by our hardware monitor. The main idea behind our monitor system is to monitor the instruction execution flow by the packet processor and detect any deviation. In order to do this, we do the following:

- Perform an offline analysis of the protocol to be executed (e.g., IP forwarding, IPSec, a firewall service) and apply the concept of software diversity generating N different binary versions.
- Generate the corresponding N compact monitoring graphs which helps the monitor in observing the core's operation
- Generate a SHA-256 hash representation for each of the N versions of the code
- Store the value of the SHA-256 hash at the beginning of the monitoring graph

The latter hash value will be used in order to perform code integrity check to ensure a secure version of the code is being loaded.

4.3 Real-time monitoring

The network processor systems need to detect attacks or failures and recover from them in order to guarantee its operation. The hardware monitor we propose functions independent from the packet processor but in parallel with it. The monitor uses as few as possible separate hardware resources. This guarantees that an attack targeting the processor will not affect the security monitor's operation and that the monitoring speed remains synchronized with the packet processing speed.

At the early stage of the runtime phase, only one of the binaries generated during the offline analysis is loaded on the network processor's instruction memory while its corresponding monitoring graph is stored in the monitor's memory. During runtime, the hardware monitor checks the flow of instructions being executed by the processor. If at any time it detects an abnormal

```

.... Common Code

158:      244704a4    addiu a3,v0,1188
15c:      35cb0014    ori   t3,t6,0x14

.... Common Code

188:      24700310    addiu s0,v1,784
18c:      35d80024    ori   t8,t6,0x24
190:      35d90028    ori   t9,t6,0x28
194:      35cd0020    ori   t5,t6,0x20
198:      340ffeff    li    t7,0xfeff

```

Fig. 5 Software diversity illustration: different instruction sequences for the same protocol - version 1 [28]

```

.... Common Code
158:      24470498   addiu  a3,v0,1176
15c:      35cd0014   ori    t5,t6,0x14

.... Common Code
188:      24700304   addiu  s0,v1,772
18c:      35d90024   ori    t9,t6,0x24
190:      35cf0028   ori    t7,t6,0x28
194:      35cc0020   ori    t4,t6,0x20
198:      3418feff   li     t8,0xfeff

```

Fig. 6 Software diversity illustration: different instruction sequences for the same protocol - version 2 [28]

operation of the processor, the processing is terminated, and a recovery process is triggered.

4.4 Recovery mechanism

In the case of an attack detection, the recovery process in the scenario of network processors is easy. It can be achieved by dropping the packets that caused the failure. This can be done without any violation because the packet delivery in the Internet protocol (IP) networks is not guaranteed. Most of the network applications are stateless, and thus, dropping partially processed packets does not lead to any inconsistencies.

Since attacks are typically based on changes in program execution, it is necessary to reset the processing stack and instruction memory so that any potential vulnerable code is overwritten. Thus, the recovery process includes the following steps:

- The packet buffer where the current packet is stored is cleared and the packet is dropped.
- The processor core that is processing the offending packet is reset: the processor stack and registers are reset to recover from any tampering with the stack pointer.
- The instruction memory is reset so that potentially harmful code is overwritten and does not affect future packets to be processed on this core.
- The control block of the N-to-1 MUX chooses at random which version of the processing binary will be reinstalled in the processor's instruction memory and which is the corresponding monitoring graph. The protocol processing code is reloaded from on-chip memory, which contains the instruction memory initialization values. That storage place is

assumed to be secure and not accessible by the attacker.

- The recovery process continues further by loading the new version of the code and its monitoring graph.

During the code loading phase, a hash value of the code being loaded is generated and compared with that found in the monitoring graph to check the integrity of the code.

Once the recovery process is completed, the processor resumes its execution. An important aspect of this recovery process is that it happens quickly and ensures that the possibility of a denial-of-service attack by sending the same attack packet is eliminated. In terms of clock cycles in our prototype implementation, the recovery process utilizes 12 clock cycles.

4.5 Content-addressable memory (CAM)

CAMs in network processing are typically used for applications such as caching, address lookup, filtering, data compression, packet classification, and various other lookup functions [30]. They are frequently used instead of algorithms executed on processors searching RAM. Traditional RAM is composed of simple storage cells, but each individual memory cell in a fully parallel CAM implementation requires embedded matching circuitry. A standard binary CAM is the simplest type of CAM. These use data search words comprised of only 1 and 0 bits, and thus, only an exact match of the search data to be performed. Ternary CAMs (TCAMs) include a third "don't care state" denoted by an X. When set locally, this comes at an additional cost, as the internal memory cell must encode three possible states instead of two. Such

an arrangement can be used to find a longest matching prefix, for example, as used in IP address lookup.

We use the TCAM memory in order to perform fast matching on the instruction being executed by the processor to determine the correct secure flow of instructions to follow.

4.6 Compact monitoring graph architecture

The monitoring graphs are simple state machines where each state represents a specific instruction. They are derived from an offline analysis of the processing code binaries. During this analysis, the binaries are divided into basic blocks. The first and final instructions of each basic block are extracted and are used by the hardware monitor to verify that the instructions executed on the processor in real time are correct. The monitoring graphs are kept relatively small using compact nondeterministic finite automata (NFA) as building blocks of their state machines.

The hardware implementation of the monitor consists of a ternary content-addressable memory (TCAM) accompanied with a Block RAM memory as shown in Fig. 7. We use the TCAM to store the final instruction of each basic block which is usually a branch instruction and the BRAM to store the branch target. The TCAM performs content matching rather than the address matching performed by standard memory cores. The content matching approach enables faster data searches than can be achieved by sequentially checking each address location in a standard memory for a particular value. The higher speed searches are achieved by using content values as an index into a database of address values. The additional ability to perform content compares in parallel enables even higher speed searches.

4.7 Monitor operation

During the operation, the current instruction being executed by the packet processor is fed to the monitor, specifically to the TCAM memory inside the monitor. The latter performs a fast search, matching its contents with the current instruction; if there is a match, the match address is provided as an index to the BRAM memory. The BRAM memory performs a read operation returning the next instruction to be executed, that is the branch target. Meanwhile, the current instruction is also stalled using a FIFO buffer for additional two clock cycles where it is compared with the returned branch address. If they match, then this is a benign instruction and there is no malicious activity. However, if the comparator finds a mismatch and the next instruction is different from the one expected, this automatically means that a malicious activity is being executed. In the latter scenario, the monitor sends a reset signal to the packet processor which performs the recovery procedure.

5 Evaluation

In this section, we evaluate our design in two different scenarios: (i) an attack scenario in which the processor deviates from the expected behavior (i.e., as a result of code injection attack or stack-smashing attack) and thus will be detected by our monitor and (ii) an instruction memory modification scenario (due to faults resulting from radiation or an attack targeting instruction memory) which will be detected during the loading phase through the integrity checking mechanism.

We present the evaluation results from a prototype implementation based on a 1st Generation NetFPGA [31] which includes a Virtex II-Pro FPGA and is used for experimental purposes. We also implement a 32-bit synthesizable MIPS Plasma processor [32] to run the

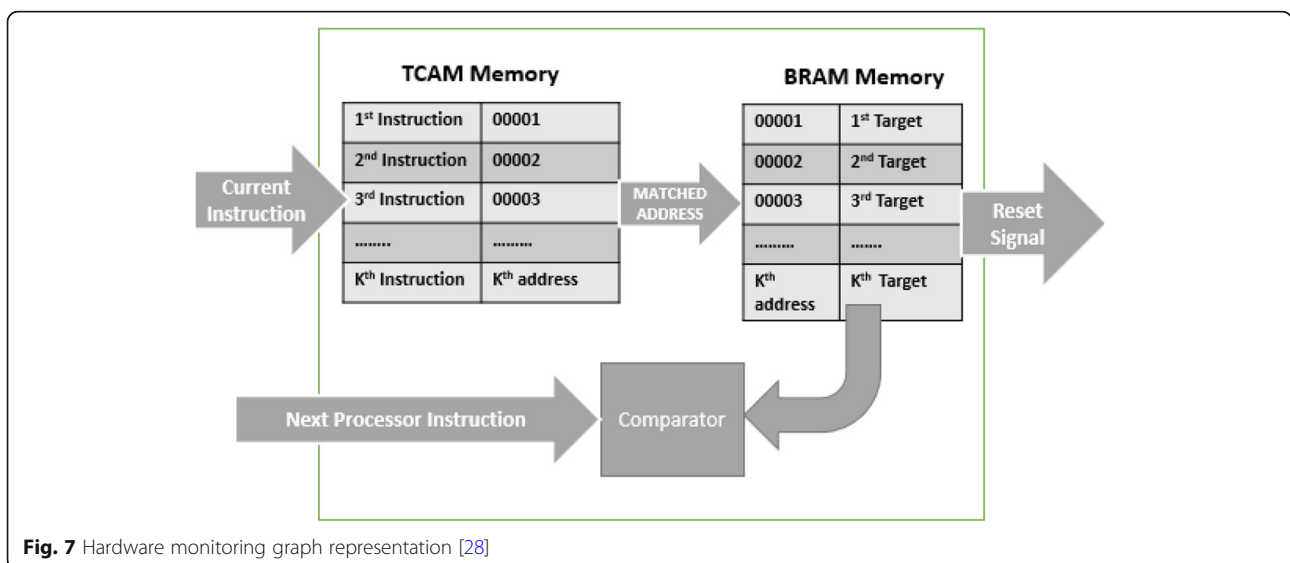


Fig. 7 Hardware monitoring graph representation [28]

protocols as a prototype proof of concept. The first task is to decide the format of the continuous monitoring stream between the packet processor and the monitor. Several options exist to monitor the execution path of the processing which include [7] the following:

- **Opcode:** By sending to the monitor opcode information, we can monitor the operations performed on the processor, which indicate the functionality of the executed application. An attacker should change the whole set of opcodes to successfully execute an attack. This is then easily detected.
- **Instruction address:** The instruction address is the unique memory address used to store the instruction set. Since this is unique, it can be used to verify the flow of instructions being executed. An attacker needs to install his malicious code in the exact memory addresses to deceive the monitor.
- **Instruction address + instruction word:** The concatenation of both the instruction address and the instruction word increases the uniqueness and ensures the fact that they can be used in the monitoring process.
- **Hash of any of the above:** Due to the hash properties, hashing any of the above results in a unique value that can also be used in monitor.

For our prototype implementation, we choose the instruction address as the format of the monitoring stream because such a pattern uses less memory resources while making it hard for an attacker to come up with an attack code.

5.1 Attack detection scenario

In the attack detection scenario, we choose to implement the IP-forwarding protocol to verify the functionality of our design; we note though that any protocol can work following the same procedure. Before we load the processing code to the packet processor, we perform off-line analysis applying the software diversity concept and generating different versions of the code. We then break each binary file into basic blocks extracting each branch origin and target to build-up our monitoring graph. We later calculate the SHA-256 hash value of each binary file and store it in the monitoring graph to verify the code during loading (this will be demonstrated in the following section).

If the application code loaded at some point executes a branch instruction, two possible outcomes exist:

- The branch is taken, and the next instruction to be executed is the branch target.

- The branch is not taken, and the next instruction to be executed is that at address $pc + 4$.

We implement the monitor to consider both benign. However, if a different outcome results, then it is detected by the monitor as a malicious activity, and thus, a recovery action is triggered.

Figure 8 represents the application disassembled code. This code should execute a branch at memory address 0x38. The branch target if taken should be 0x30; otherwise, the next instruction to be executed is at address 0x3c. Figure 9 represents the monitoring graph we prepared for the benign nonmalicious code. For the monitoring graph, the right branch target is 0x3f instead of 0x30 if the branch is taken. Figure 10 shows the operation of the whole system (the packet processor and the monitor). This figure shows that when the instruction 0x38 is being executed, it is directed to the TCAM element of the monitor at the same time. It takes two clock cycles for the TCAM memory to find a match and a matched address is ready. This matched address is provided as an index to the BRAM memory which returns the next address that is the branch target. The latter is compared with the instruction being executed by the processor taking into consideration the number of cycles already passed. In this case, the monitor expects an instruction 0x3f whereas the instruction being executed is 0x34.

The monitor detects this malicious activity and initiates the recovery procedure by sending a reset signal to the packet processor. The operation as a whole is as follows: the detection of the malicious activity occurs after three clock cycles, and the recovery process is triggered at the fourth cycle.

5.2 Integrity checking mechanism scenario

In this scenario, we demonstrate the integrity checking mechanism in our design. The main purpose of this mechanism is to insure the code integrity, that is, the code being loaded has not been modified. This mechanism will happen during the code loading phase.

Figures 11 and 12 show the hash calculation during the loading phase of the binary code. However, Fig. 11 shows the loading of a code that has not been modified or tampered with or even faulty. The hash calculated was as the expected hash value stored in the monitoring graph. Therefore, the processor and the monitor will proceed with the expected operations. On the other hand, Fig. 12 shows a case where the calculated hash of the loaded code is different from the expected hash stored in the monitoring graph and that is why the reset signal was initiated. Hence, Fig. 12 proves how our integrity checking mechanism can detect whether the code has been modified or has been tampered with. It is

```

Application Code

00000000 <__start>:
0:      10000003   b 10 <__start+0x10>
4:      00000000   nop
8:      10000012   b 54 <__start+0x54>
c:      00000000   nop
..... Some Processing Code
30:     ac800000   sw    zero,0(a0)
34:     0085182a   slt   v1,a0,a1
38:     1460ffff   bnez  v1,30<__start+0x30>
3c:     24840004   addiu a0,a0,4
40:     34080001   li    t0,0x1

```

Fig. 8 Disassembled application code [28]

worth noting that code modifications can be the result of a bit flip in the storage or the result of an attack.

5.3 Resource utilization

The C-based implementation of the protocol used in this experiment produces a 16 K bits binary file. This size is considered adequate for simple implementation of popular routing services. Table 1 shows the resource consumption of our design on the NetFPGA. In terms of memory, an average processing binary file utilizes three BRAM16 blocks, where each BRAM block is equivalent to 18 K bits of memory. This represents a very small percentage of 1.3% of the available BRAM-16 blocks in the NetFPGA-1G platform. If the same design was ported to the NetFPGA-10G platform, which contains a Vertex-V FPGA, each processing binary file will take up to 0.46% of the available memory. The latter resource utilization shows that we can easily afford to design multiple versions of the same program without exhausting the available resources.

Monitoring Graph

```

0x10
0x54
0x3f

```

Fig. 9 Monitoring graph entries [28]

In terms of TCAM memory requirements, one sample-monitoring graph corresponding to the previously produced binary file has a size < 1 kbit (480 bits to be specific). For the current TCAM resources available in today's routers (18 Mbits [33]), such monitoring graph will utilize < 0.5% of the available resources. Therefore, the performance overhead of our design is not a burden on the available TCAM resources in today's modern routers.

In terms of speed and throughput, we are limited by the capabilities of the FPGA. It runs at 62.5 MHz, but even at such a low clock rate, we can achieve an average throughput of 64.1 Mbps. While interpreting the results, we should also consider that we experiment with the IP-forwarding protocol over small packets. The rates may increase with the increase of the size of the packets being forwarded.

In terms of LUT slices, our design utilized 278 LUT slices for the monitor implementation and 2284 LUT slices for the monitor and processor combined. Additionally, the SHA-256 hashing module utilized 639 LUT slices. These resources are considered adequate and efficient on the FPGA used for the prototype implementation.

5.4 Experimental comparison with state of the art techniques

In this section, we present an experimental comparison between our design and two other state of the art techniques existing in literature: the "IMPRES: Integrated Monitoring for Processor Reliability and Security" design [21] and the "Secure Embedded Processing through Hardware-Assisted Run-Time Monitoring" design [6].

In [21], the authors presented the Integrated Monitoring for Processor Reliability and Security technique

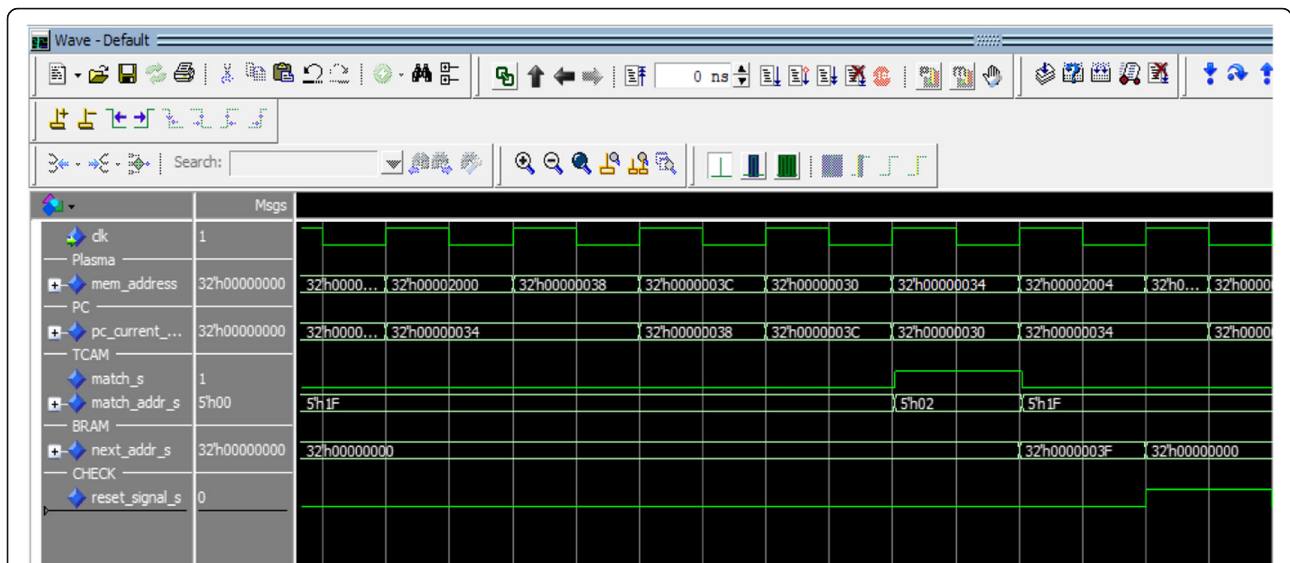


Fig. 10 Simulation results [28]

which is a hardware/software technique that acts at the granularity of the micro-instructions. They proposed such technique to detect code injection attacks and bit flips in instruction memory. In their design, they divide the binary code into different basic blocks, evaluate a checksum for each block, encrypt the checksum with a secret hardware key during loading, and recalculate the

encrypted checksums during runtime. In order to implement their design, they had to modify the binary code adding a “chk” instruction at the beginning of each basic block that carries the encrypted checksum for the corresponding basic block. Unlike the approach in [21], the design we are proposing does not modify the binary code by any means. We perform an offline analysis and

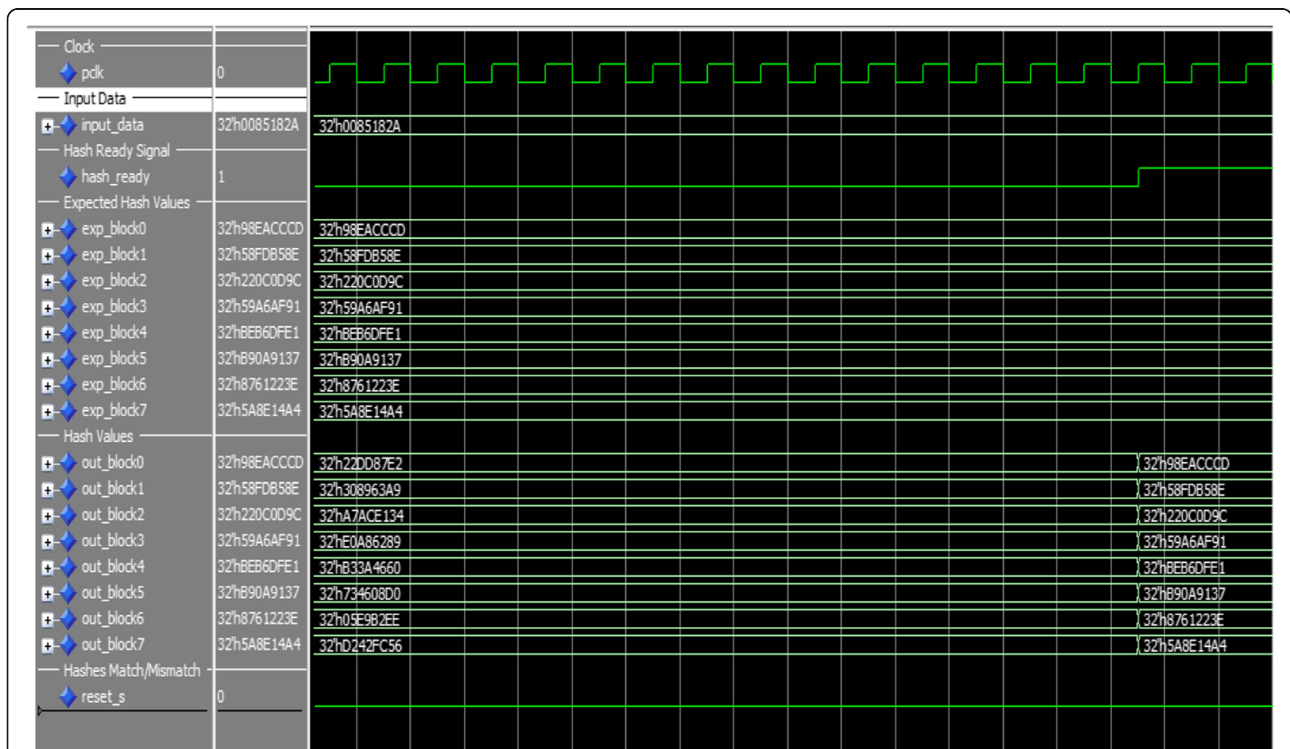


Fig. 11 Integrity checking: hash calculation of a benign code

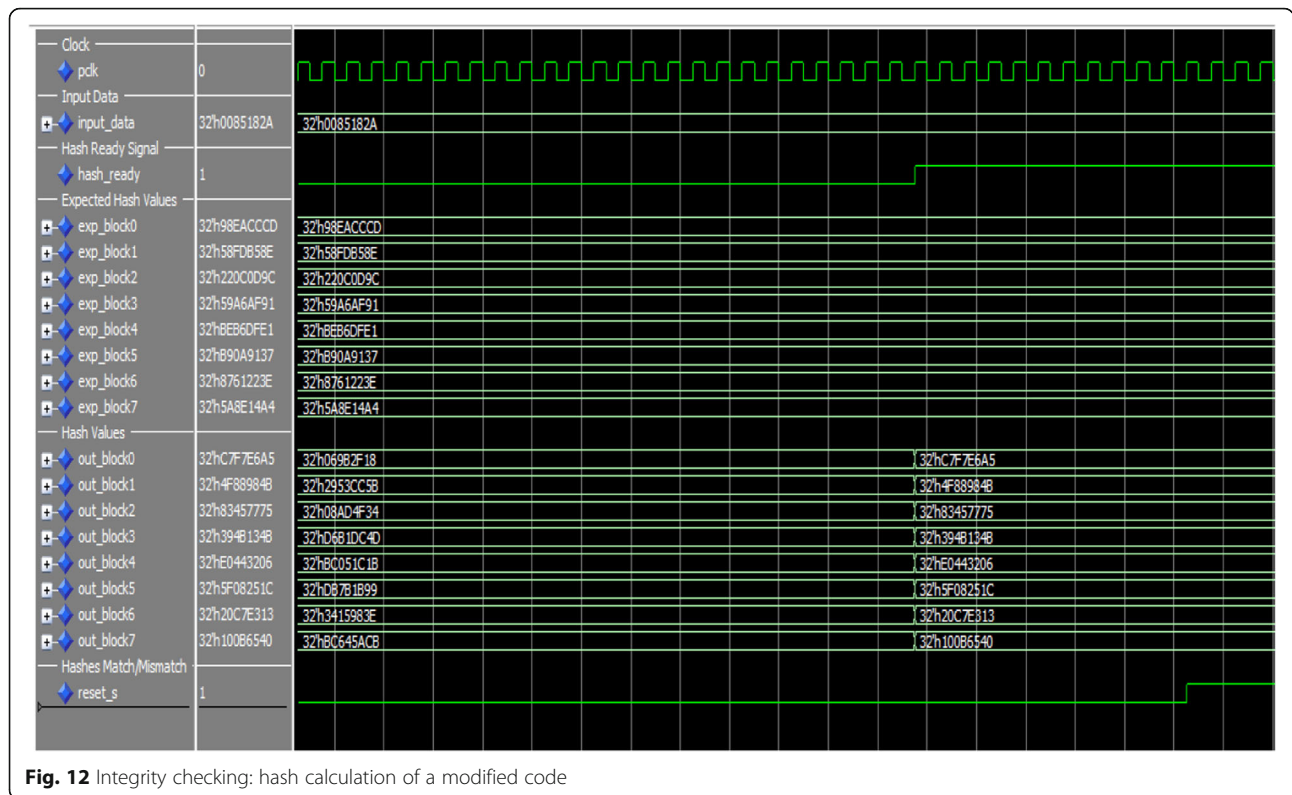


Fig. 12 Integrity checking: hash calculation of a modified code

extract a compact monitoring graph which corresponds to the first and final instruction of each basic block. Additionally, we only perform the hash evaluation once, i.e., during the secure loading phase of the code which also saves in memory, time, and performance.

Furthermore, in [21], the authors do not implement any approach to address the existence of the vulnerability in the first place. They are able to detect it, but it is still there. An attacker can thus persist on resending the attack packet over and over leading to a denial of service for legitimate processing tasks. However, in our approach, we implement the concept of software diversity which replaces the vulnerable version of the code with a new secure version and thus eliminating the possibility of the aforementioned attack.

In [6], the authors presented a hardware-assisted paradigm to enhance the embedded system security by detecting and preventing unintended program behavior. In their work, the authors presented an architecture where the processor is augmented with a hardware monitor

that monitors the processor’s dynamic execution trace, checks whether it falls within the allowed program behavior, and flags any deviation. Additionally, the authors identified certain properties that can be used to define permissible program behavior. These properties include the inter-procedural control flow of the program, the intra-procedural control flow for each function, and the integrity of the instruction stream. In order to implement the inter-procedural control flow checking mechanism, they had to utilize two look-up tables that store the function start and return addresses. Each look-up table was implemented using a TCAM memory, thus using an expensive resource in the available routers. Additionally, to implement the intra-procedural control flow checker, they utilize a basic block table where each entry consists of additional subentries and thus utilizing more memory from the available resources. Furthermore, their integrity checking mechanism is based on basic block hash calculation where during program execution, the monitor buffers the instruction stream corresponding to each basic block until a jump/branch instruction is encountered, after which happens the hash calculation and comparison. If the buffer becomes full at any time, the processor is stalled in order to allow the integrity checking mechanism to catch up. Finally, in their design and after extracting the aforementioned information, the extracted information is translated into data and appended to the program’s binary code.

Table 1 Resource consumption and performance of the monitoring graph

	Our design	NetFPGA-1G	NetFPGA - 10G
BRAM-16	1	232	648
Speed (MHz)	62.5	62.5	550
Throughput (avg in Mbps)	64.1	N/A	N/A

Unlike the design presented in [6], we do not modify the binary code in the design we are proposing neither by adding additional instructions nor by appending more information to the data section. In our design, we perform a similar approach to the inter-procedural and intra-procedural control flow checking through one step, thus utilizing less look-up tables (less TCAM memory) and less BRAM memory for the basic blocks (check Fig. 13) saving on memory resources. Additionally, the hash calculation in our design is done during the loading phase only, not the runtime phase, and thus, the processor will never be stalled for the hash calculation. Furthermore, storing the hash values with the data being hashed together is a security weakness that can be a target for a smart attacker. Finally, in [6], once the vulnerability is detected, it is still there and can be used for a denial of service attack; whereas in our design, we utilize the software diversity to load a new secure version of the code and thus preventing such types of attack.

Figure 13 shows a detailed comparison between our design and the aforementioned approaches when running three different protocols: CM-protocol (a protocol

written by us for experimental purposes), IP-forwarding, and IP-Sec protocols. The comparison is in terms of four main categories: additional clock cycles required, binary code size (lines of code), TCAM memory required resources, and BRAM memory required resources.

The efficiency of our design is evident from Fig. 13. Both designs [6, 21] require additional amount of clock cycles to perform their monitoring functionality. This is because in both designs, the monitors perform runtime hash calculation for each basic block, unlike our design which checks the integrity of the code during loading phase only. Considering the additional lines of code, the IMPRESS design [21] has a higher percentage than the hardware-assisted runtime monitoring approach because additional “chk” instructions are added for each basic block. On the other hand, the hardware-assisted runtime monitoring approach rarely adds a “jmp” instruction when necessary. In contrast to both designs, our approach does not modify the binary code at all. Additionally, when considering the memory resources required, the hardware-assisted runtime monitoring design will always require more TCAM resources than our design

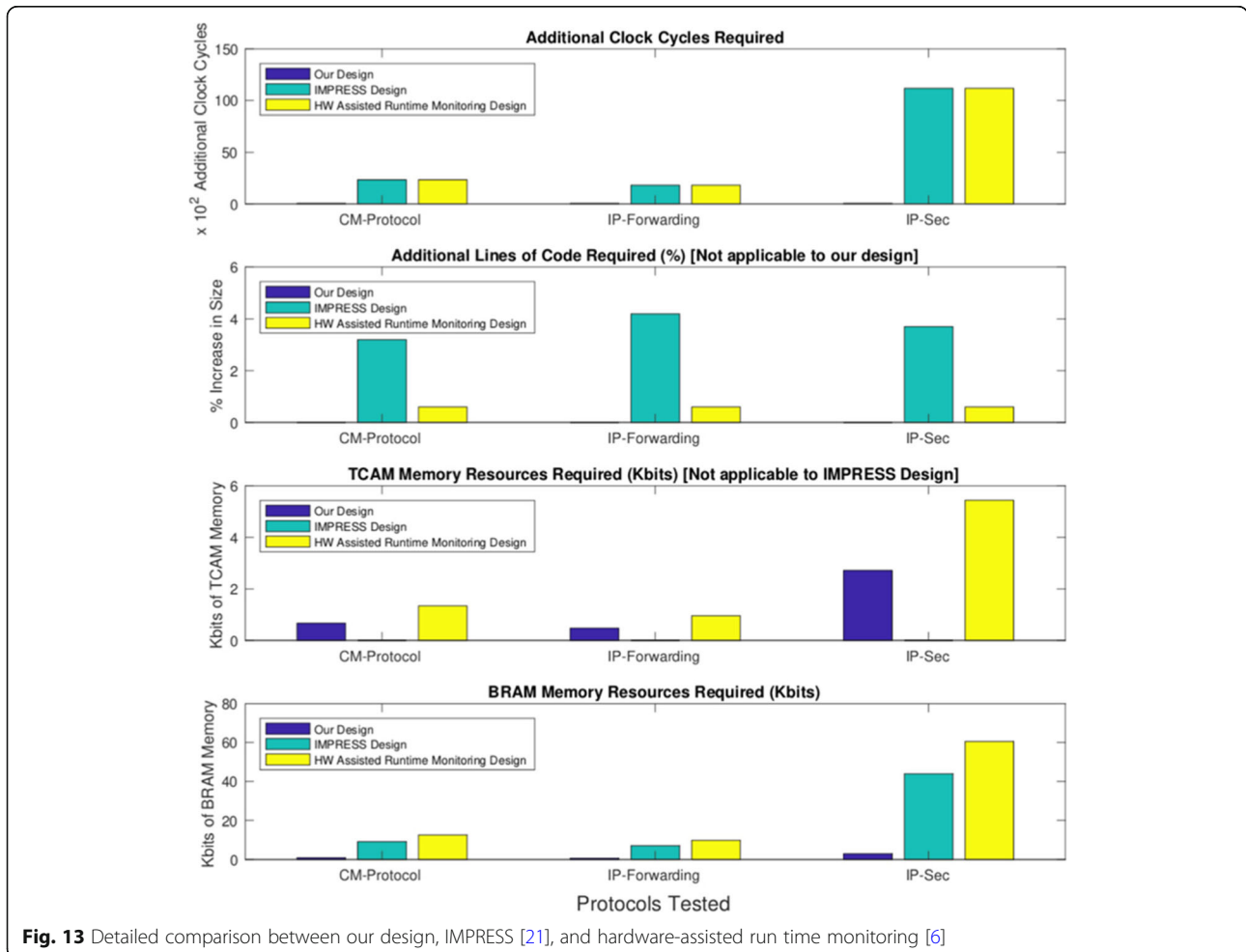


Fig. 13 Detailed comparison between our design, IMPRESS [21], and hardware-assisted run time monitoring [6]

due to the existence of two lookup tables. Furthermore, considering the BRAM memory resources required, both designs require more memory resources than our design because they store more information about each basic block and its corresponding hash. Unlike both designs, we only store one hash value and the target of branches extracted from the basic blocks. Hence, our design outperforms the state-of-the-art monitoring techniques found in literature.

6 Conclusion

The use of routers equipped with general-purpose processing engines and executing software-based packet processing is becoming more prevalent in the Internet. Using software-based processing in the data plane of the network presents a target for novel intrusion and denial-of-service attacks. This can have a significant impact on the overall security of the network. In our work, we present the design of an adaptive security mechanism for modern packet processors. We present a prototype implementation of a secure monitoring system which can be used in order to detect the attacks targeting the data plane of the network. This monitoring system is fault tolerant and reliable taking advantage of software design diversity comparing the operation of the processing cores to its corresponding expected behavior obtained by the offline analysis of the packet processing binary. The monitoring system will include different behaviors corresponding to different executions of the binary to be processed. The system continuously checks the execution of the processor and triggers a recovery mechanism if a deviation from the behavioral execution path is detected. The monitoring system we present is fast because it implements a new type of memory, the TCAM memory, performing fast searching and matching within two clock cycles. It is also a secure monitoring system implementing a SHA-256 hash which ensures the integrity of the binary code loaded. The prototype implementation of our monitoring system shows that our system is an effective approach in protecting the networking infrastructure in the future Internet with a negligible additional resource utilization and a very good speed.

Abbreviations

ASICs: Application-specific integrated circuits; FIFO: First In First Out; FPGA: Field Programmable Gate Array; MPSoC: Multiprocessor systems-on-chip; NFA: Nondeterministic finite automata; RAM: Random access memory; SDN: Software-defined networking; TCAM: Ternary Content-Addressable Memory

Acknowledgements

Not applicable

Funding

This research was not supported by any external funding source.

Availability of data and materials

Please contact authors for data requests.

Authors' contributions

DC conceived of the study and its design and coordinated and helped to draft the manuscript. CM implemented the design, performed the experimental work, and drafted the paper. Both authors read and approved the final manuscript.

Authors' information

Not applicable

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 24 April 2018 Accepted: 27 December 2018

Published online: 10 January 2019

References

- Jacobson, V., Smetters, D. K., Thornton, J. D., Plass, M. F., Briggs, N. H., & Braynard, R. L. (2009). Networking named content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies* (pp. 1–12). ACM. <https://dl.acm.org/citation.cfm?id=1658941>
- Kirkpatrick, K. (2013). Software-defined networking. *Communications of the ACM*, 56(9), 16–19.
- Anderson, T., Peterson, L., Shenker, S., & Turner, J. (2005). Overcoming the Internet impasse through virtualization. *Computer*, 38(4), 34–41.
- Mogul, J. C. (1989). *Simple and flexible datagram access controls for...* (In USENIX Conference Proceedings). Citeseer. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.187.9099>
- Chasaki, D., Wu, Q., & Wolf, T. (2011). Attacks on network infrastructure. In *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)* (pp. 1–8). IEEE. <https://ieeexplore.ieee.org/abstract/document/6005919>
- Arora, D., Ravi, S., Raghunathan, A., & Jha, N. K. (2005). *Secure embedded processing through hardware-assisted run-time monitoring* (pp. 178–183). Munich: Proc. of the design, automation and test in Europe conference and exhibition (DATE'05).
- Chasaki, D., & Wolf, T. (2013). *Attacks and defenses in the data plane of networks*. IEEE Transactions on Dependable and Secure Computing. <https://ieeexplore.ieee.org/document/6231636>
- Standard, D. E. (2002). *Federal Information Processing Standards Publication (FIPS PUB) 180-2*, National Institute of Standards and Technology (NIST).
- Kent, S., & Atkinson, R. (1998). *Security architecture for the Internet protocol. RFC 2401*.
- Estevez-Tapiador, J. M., Garcia-Teodoro, P., & Diaz-Verdejo, J. E. (2004). Anomaly detection methods in wired networks: a survey and taxonomy. *Computer Communications*, 27(16), 1569–1584.
- Savage, S., Wetherall, D., Karlin, A., & Anderson, T. (2001). Network support for IP traceback. *IEEE/ACM Transactions on Networking (TON)*, 9(3), 226–237.
- Caswell, B., & Roesch, M. (2004). *Snort: the open source network intrusion detection system*.
- Cui, A., Song, Y., Prabhu, P. V., & Stolfo, S. J. (2009). Brave new world: pervasive insecurity of embedded network devices. In *Recent Advances in Intrusion Detection* (pp. 378–380). Springer. <https://www.springer.com/us/book/9783642043413>
- Parameswaran, S., & Wolf, T. (2008). Embedded systems security an overview. *Design Automation for Embedded Systems*, 12(3), 173–183.
- Tokuda, H., Kotera, M., & Mercer, C. E. (1988). *A real-time monitor for a distributed real-time operating system* (Vol. 24). ACM. <https://dl.acm.org/citation.cfm?id=69222>
- Dhawan, M., Poddar, R., Mahajan, K., & Mann, V. (2015). *Sphinx: detecting security attacks in software-defined networks*. NDSS. <https://dblp.uni-trier.de/rec/bibtex/conf/ndss/DhawanPMM15>
- Braga, R., Mota, E., & Passito, A. (2010). Lightweight DDoS flooding attack detection using NOX/OpenFlow. In *Local Computer Networks (LCN), 2010 IEEE 35th Conference On* (pp. 408–415). IEEE. <https://ieeexplore.ieee.org/document/5735752>
- Wang, Y., Zhang, Y., Singh, V., Lumezanu, C., & Jiang, G. (2013). Netfuse: short-circuiting traffic surges in the cloud. In *Communications (ICC), 2013*

- IEEE International Conference On* (pp. 3514–3518). IEEE. <https://ieeexplore.ieee.org/document/6655095>
19. Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., & Shenker, S. (2008). NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3), 105–110.
 20. Mao, S., & Wolf, T. (2007). Hardware support for secure processing in embedded systems. In *Proceedings of the 44th Annual Design Automation Conference* (pp. 483–488). ACM. <https://dl.acm.org/citation.cfm?id=1278605>
 21. Ragel, R. G., & Parameswaran, S. (2006). IMPRES: integrated monitoring for processor reliability and security. In *Proceedings of the 43rd Annual Design Automation Conference* (pp. 502–505). ACM. <https://ieeexplore.ieee.org/document/1688849>
 22. Ragel, R. G., Parameswaran, S., & Kia, S. M. (2005). Micro embedded monitoring for security in application specific instruction-set processors. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (pp. 304–314). ACM. <https://dl.acm.org/citation.cfm?id=1086337>
 23. Nakka, N., Kalbarczyk, Z., Iyer, R. K., & Xu, J. (2004). An architectural framework for providing reliability and security support. In *2004 International Conference on Dependable Systems and Networks* (pp. 585–594). IEEE. <https://ieeexplore.ieee.org/document/1311929>
 24. Zambreno, J., Choudhary, A., Simha, R., Narahari, B., & Memon, N. (2005). SAFE-OPS: an approach to embedded software security. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(1), 189–210.
 25. Chen, X., Chasaki, D., & Wolf, T. (2013). External monitoring of highly parallel network processors. In *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)* (pp. 197–204). IEEE. <https://ieeexplore.ieee.org/document/6602312>
 26. Mansour, C., & Chasaki, D. (2016). Power monitoring of highly parallel network processors. In *2016 International Conference on Computing, Networking and Communications (ICNC)* (pp. 1–5). IEEE. <https://ieeexplore.ieee.org/document/7440713>
 27. Mansour, C., El Hajj Shehadeh, Y., Chasaki, D.: Design of an adaptive security mechanism for modern routers. In: 2015 IEEE International Conference on Consumer Electronics (ICCE), pp. 241–244 (2015). IEEE. <https://ieeexplore.ieee.org/document/7066397>
 28. Mansour, C., & Chasaki, D. (2016). Trust and reliability for next-generation routers. In *MILCOM 2016–2016 IEEE military communications conference* (pp. 740–745). <https://doi.org/10.1109/MILCOM.2016.7795417>.
 29. Chen, L., & Avizienis, A. (1978). N-version programming: a fault-tolerance approach to reliability of software operation. In *Digest of papers FTCS-8: Eighth annual international conference on fault tolerant computing* (pp. 3–9).
 30. McLaughlin, K., O'Connor, N., & Sezer, S. (2006). Exploring cam design for network processing using fpga technology. In *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)* (pp. 84–84). <https://doi.org/10.1109/AICT-ICIW.2006.96>.
 31. Lockwood, J. W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., & Luo, J. (2007). NetFPGA—an open platform for gigabit-rate network switching and routing. In *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education, San Diego, CA* (pp. 160–161).
 32. Plasma Processor, <https://opencores.org/projects/plasma>
 33. Pagiamtzis, K., & Sheikholeslami, A. (2006). Content-addressable memory (cam) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3), 712–727.

Submit your manuscript to a SpringerOpen[®] journal and benefit from:

- Convenient online submission
- Rigorous peer review
- Open access: articles freely available online
- High visibility within the field
- Retaining the copyright to your article

Submit your next manuscript at ► [springeropen.com](https://www.springeropen.com)
