Towards Dynamic Execution Environment for System Security Protection against Hardware Flaws

Kenneth Schmitz[†] Oliver Keszocze[*†] Jurij Schmidt[*†] Daniel Große[*†] Rolf Drechsler[*†]

[*]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[†]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{kenneth, keszocze, grosse, drechsler}@cs.uni-bremen.de

***Abstract*—Attacks exploiting security flaws in software are very common. They are typically addressed during the ongoing software development process or by providing software patches. Attacks making use of hardware related flaws via malicious software recently gained popularity. Prominent examples are errata-based, aging-related or, for example, the infamous Rowhammer-attack. In this paper, we present an approach to detect software-based attacks which exploit hardware flaws. Since the flaws are typically triggered by characteristic instruction sequences, our approach is implemented as a dynamic execution environment for program monitoring at runtime. Several case studies underline the effectiveness and the low overhead of our approach.**

I. Introduction

Malicious software such as Trojans or viruses can be accounted for major system failures and large financial losses [1]. Most recently, cryptographic ransomware was used for blackmailing companies to recover their encrypted data [2]. To protect the victim's systems against such attacks, typically several techniques (e.g. sandboxing, static/dynamic or signature based analysis) have been implemented in antivirus software. While there are different arguments for and against each of these techniques, they have been developed from a software centric perspective since malicious code uses flaws and vulnerabilities in software as an attack vector.

Due to the shrinking feature sizes and the increasing complexity of hardware, more flaws reach the silicon [3]. Hence, focusing on hardware and potential attacks exploiting the flaws in silicon is very important and mandatory. In recent years, many approaches for Trojan/backdoor identification in hardware and *Integrated Circuit* (IC) counterfeit detection [4] have been developed. Furthermore, defined areas for secure software execution and data storage in hardware (e.g. ARM TrustZone, Intel SGX, TPM) have been created, aiming for the protection of data and the software itself during execution. In contrast, this work takes the hardware perspective and aims to protect systems against malicious software which exploits *hardware flaws* as a new attack vector. In the following we identify two major categories of hardware flaws where an urgent protection is inevitable.

Errata-based defects and resulting system failures are the first category. Almost every processor-generation has errata instructions, which are typically addressed by microcode- or BIOS-updates. Since modern hardware components are very complex, verification and test become more challenging and flaws can remain undiscovered prior to the fabrication. Powerful instruction set extensions to the x86 *Instruction Set Architecture* (ISA) have been recently reported to result in unpredictable behavior [5]. Undocumented features inside the ISA, which *can* cause unpredictable system behavior, have been revealed [6] as well.

The second category covers flaws which are inherited from the feature sizes used to fabricate the components. The Rowhammer-attack affects *Random Access Memory* (RAM) and *Solid-State Drives* (SSDs) [7]. The *malicious aging in circuits/cores* (MAGIC) [8] leads to very fast semiconductor aging. *Field Programmable Gate Array* (FPGA)-based systems are susceptible to this attack scenario [9] as well. Both attacks exploit the basic properties of the feature sizes in order to make the system fail early or unexpectedly. All of these flaws can be induced by regular execution of regular software. Unfortunately, antivirus software typically fails if confronted with scenarios, which explicitly target hardware flaws.

In this paper, we propose a novel hardware-centric approach based on the following idea: All of the presented attacks exploit the hardware flaws through characteristic instruction sequences. Since detecting these instructions sequences is possible at the instruction level, we present an engine for instruction-screening. We use the *Quick Emulator* (QEMU) [10], providing a code translation layer that grants access to single instructions during execution. Our approach detects *all* user-defined instruction (search-) patterns in the instruction sequence of the executed program.

Unfortunately, a given platform can be susceptible to more than one attack. Therefore, the approach must allow for searching of *all* instructions at the same time. In addition, it must detect spatially distributed instruction sequences in programs to protect against advanced attacks (e.g. clflush-based Rowhammer-attacks) and by this providing a general scheme in contrast to existing solutions. As a consequence, we have identified the powerful string matching algorithm Aho-Corasick [11] serving as basis for screening the instructions. Parallel matching of search patterns, while maintaining linear complexity with respect to the input sequence plus the number of simultaneously matched search patterns, is the strength of this algorithm. To handle the above mentioned spatially distributed instruction sequences in programs, we extended Aho-Corasick to match interrupted instruction sequences while maintaining the same algorithmic complexity. In our experiments, the extended algorithm was able to cope with the

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SecRec under grant no. 16K1S0606K and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

execution speed (and the resulting high instruction-throughput) at the translation layer of QEMU at runtime.

The proposed approach also addresses software subroutines which implement malicious behavior. If the functionality of an application is known, certain instruction patterns can indicate malicious intents. Examples can be office applications *without* update functionality which contains instructions implementing network communication, or the Linux built in copy command using specific cryptographic instructions. These instructions are implausible in this specific context and can indicate an attack. The proposed approach can also recognize this behavior during runtime and intervene if the operational security is at risk.

II. Related Work

Exploiting flaws in hardware and the protection thereof is an ongoing challenge. In general, software is capable of transitioning hardware to a state from where there is no recovery but a hard system reset. Even physical, irreversible damage is possible [12]. Identifying software with malicious intents is a very complex problem and has been thoroughly investigated (for an overview see for instance [13]). It has been shown that behavioral detection, signature extraction and improving resilience to automatic mutations still fail in many scenarios and remain a major challenge [14], [15]. In the software domain regular viruses use sophisticated techniques in order to hide their true intentions. Among these are self-decryption, oligo-, poly- or metamorphism which change the appearance (in terms of instruction sequences and behavior) during or prior to execution. Attacks against hardware flaws often require specific, immutable sequences of instructions (e.g. errata-related) to trigger the erroneous behavior.

The authors of the *Micro-Architectural Side Channel Attack Trapper* (MASCAT) [16] relied on static code analysis in order to scan for microarchitectural attacks. The work focuses on fully automated off-line analysis of applications (e.g. for app stores). Thus, this approach does not provide any protection as soon as the binary has reached the target system.

For the particular example of the Rowhammer-attack, the authors of [17] identified circumstances, where the attack can escape the known `clflush`-pattern and still remain effective.

In [18], a sandboxing technique has been proposed which restricts access to memory region within the host's address space. The scheme protects the host program from reads and writes by its guests and it allows the restriction of the instruction set available to guests. Hence, a full notion of which instructions are *permissible* for every individual program is required.

There are concepts capable of repairing, patching or correcting the erroneous system behavior on the hardware level. Such a detection scheme, based on errata and internal signal observation, has been presented in [19] and [20]. Furthermore, the overall system state can be monitored and corrected on hardware level, as proposed in [21] and [22]. All of these approaches require hardware modifications which are impractical after fabrication.

III. Preliminaries

QEMU and its internal translation method as well as the the efficient Aho-Corasick string matching algorithm are the key components for our approach and are reviewed in this section.

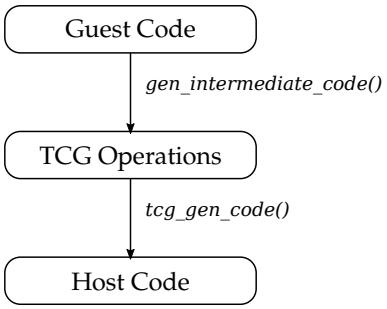

Fig. 1. Regular translation flow

A. Quick Emulator (QEMU)

QEMU is an open-source cross-platform environment for virtualization and emulation. It can utilize the *Kernel Virtual Machine* (KVM) for hardware acceleration. Two different modes of operation are provided:

A complete host system (including peripheral devices) can be mimicked to the guest system when the full virtualization environment is used. KVM is leveraged to speed up guest systems to near native execution speed and I/O redirection accelerates peripheral hardware access.

The second mode provides an user-mode emulation, which executes a single program as a guest application on a host system. When a program is run in user-mode emulation, independence from compiler versions and architectures is desired. The *Tiny Code Generator* (TCG) manages the code translation from the guest architecture to the host architecture. Guest instructions are translated to a *machine-independent intermediate notation* which is recompiled for the host's architecture. Several optimizations are applied in this step during emulation mode. All guest instructions are fully accessible at the TCG interface and can be monitored at runtime.

Figure 1 shows the relevant part of QEMU's flow, which is essential for this work and where our approach is included. All TCG operations are derived from the guest application's code. This intermediate language is processed by the TCG and translated to the target architecture. Finally, the TCG-generated target code is executed on the host's hardware.

B. The Aho-Corasick String Matching Algorithm

The Aho-Corasick algorithm is a dictionary-based string matching algorithm. It *simultaneously* locates all strings of a finite set of search strings within an input sequence. The algorithm is known to be highly scalable [23]. A dictionary is computed in advance – resulting in a tree structure called trie – to achieve the desired complexity which is linear in the length of the input plus the number of matched entries. A trie or a prefix-tree is a data structure which is typically used to store characters for search operations on character sequences. This specialized search-tree, implements storage for multiple character sequences simultaneously. The trie implicitly compresses the stored data, since shared prefixes are stored only once.

An example is shown in Figure 2. Starting from the root node, the algorithm traverses the tree while matching individual characters from the dictionary. The tree represents the entire dictionary $\{\{a\},\{a,b,a\},\{c,a,b,d\}\}$. All dotted nodes

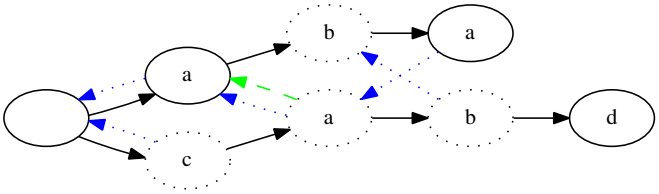

Fig. 2. Basic example for a dictionary-tree of {{a}, {a,b,a}, {c,a,b,d}}

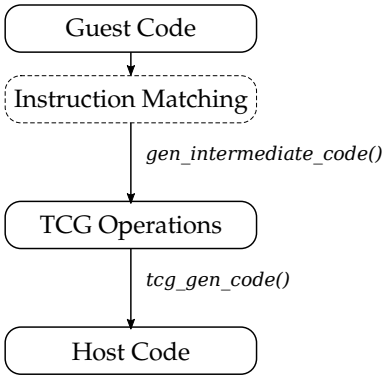

Fig. 3. Extended translation flow

are intermediate nodes. The solid nodes, in contrast, are target nodes representing a successful detection. Additional arcs have to be computed to allow fast transitions between failed string matches. Connections denoted by dotted blue arcs are called suffix arcs, which point to the longest possible strict suffix in the graph. They are computed in linear time by traversing the dotted arcs of a node's parent until the child is matching the character of the arc's target node. Connections denoted by dashed green arcs are called dictionary suffix arcs, which point to the next reachable solid node following blue arcs. These are computed in linear time as well by traversing the dotted arcs until a solid node is found.

IV. Dynamic Execution Monitoring

In this section, the implementation of our approach is presented in detail. Searching for malicious intents in software requires access to *all* program instructions. Hence, we conducted profiling experiments to determine the interface in QEMU's architecture, where the complete instruction sequence can be monitored. Figure 3 shows *where* our approach has been implemented in QEMU's architecture. The profiling experiments also revealed the TCG as the key-component for QEMU's speed. Subsequently, all extensions to this layer must be efficient in terms of their computational complexity to preserve the performance. The TCG implements the boundary after which code will be executed by the host's processor. Hence, the proposed solution detects malicious instruction sequences *prior* to translation in order to realize the system protection.

Since instruction screening is similar to searching for a matching character sequence in a string, the Aho-Corasick algorithm was chosen due to its strengths in the parallel matching of search patterns. However, in contrast to regular string matching, the suspect instruction sequences are rarely a sequence of consecutive elements. Hence, we extended the basic

```
label:                  label:
  mov (A), %eax           mov (A), %eax
  mov (B), %ebx           mov (B), %ebx
  clflush (A)             clflush (A)
  clflush (B)             clflush (B)
  jmp label               (*)
                          jmp label
```

Fig. 4. Pseudo assembly for Rowhammer-attacks

Aho-Corasick algorithm to cope with the spatial distribution of instructions inside an executable.

A. Extension of the Aho-Corasick String Matching Algorithm

The following example can motivate this necessity in a clear fashion. The loop in Figure 4 implements a clflush-based Rowhammer-attack.

The asterisk character represents an arbitrary instruction or instruction sequence within the malicious sequence. Strictly searching for the left pattern will ignore the example provided on the right, although it will yield the same effect. Hence, our approach must be able to skip intermediate instructions. In order to achieve this functionality, two major extensions were necessary:

1. The spatial distribution of instructions in an sequence has been addressed by *Don't-Care* (DC) nodes in the language of the dictionary. These nodes provide the algorithm with the capability to skip intermediate segments until the next valid instruction is found.
2. Since the performance benefit granted by efficient transitioning between failed string-matches in the dictionary-tree is essential, complex trees with an arbitrary amount of DC nodes are impractical. Hence, we implemented a partial recompilation and modification of the dictionary-tree during runtime. After a DC node is reached, the remainder after the DC node is inserted at the root node. This addresses both requirements: The ability to search for all patterns simultaneously is preserved, and the reliable detection of spatially separated instruction segments is possible. Finally, a bidirectional connection between each DC node with its associated remainder is stored. This establishes the reattachment of the remainder to its former position in linear time, when the search is completed.

The proposed method is implemented as a non-greedy pattern search procedure: After a sequence of ignored instructions (DC instructions), the first matching instruction will be interpreted as the end of the DC-sequence. Finally, the dictionary-tree will be reverted to its initial state (with respect to the active search-pattern).

B. Search-Pattern Matching

A valid search-pattern must contain all instructions necessary for a successful detection. If the pattern is present in the executed binary, the implemented solution will detect the sequence in the binary-stream during execution. However, depending on the *strength* of this search pattern, false-positives are possible, since the spatial range of the search algorithm can exceed meaningful boundaries such as methods, blocks or loop-bodies. These *false-positives* are only false in the sense, that

they will not have the intended or malicious effect. Nevertheless, the implementation will only report the presence of a search pattern if the given instruction sequence is actually present in the binary. In contrast, if the provided search pattern is fully specified (no DCs), false-positives can not occur.

In the following, a compact example will clarify the search-pattern matching procedure.

Figure 5 provides an example of a initial dictionary-tree. It shows the computed tree for the following dictionary: $\{\{i_1, i_2\}, \{i_1, *, i_3\}, \{i_0, *, i_5\}\}$. These sequences can be characterized and numbered as follows.

1. $\{i_1, i_2\}$ – Instruction i_1 is directly followed by i_2 for a successful detection.
2. $\{i_1, *, i_3\}$ – Instruction i_1 followed by an arbitrary number of instructions until instruction i_3 is found.
3. $\{i_0, *, i_5\}$– Instruction i_0 followed by an arbitrary number of instructions until instruction i_5 is found.

This compact tree representation stores all active search-patterns. Each instruction from the instruction-stream will be matched with either the root-node or with the following node in case of an active detection. After a DC node is reached, the dictionary-tree is altered. Figure 6 shows the temporal insertion of the instruction sequence's remainder at root level after the detachment from the DC node according to sequence 2. This way, an arbitrary amount of instructions can be skipped until the remainder is matched and the search-pattern is completed. Figure 7 shows the inserted remainder at root level according to sequence 3. In order to maintain low computational complexity, the dotted arcs indicate the connection for reattachment after completion of a search-pattern.

The resulting matching procedure is shown in Figure 8. Since all sequences start with i_0 or i_1, they will be compared with each element of the input instruction sequence from the executed program. This is reflected in the comparison of *all* outgoing arcs from the root node of the initial dictionary-tree. After matching i_0 from seq. 3 and i_1 from seq. 2, the DC nodes are expanded and the algorithm compares the following instructions with instruction i_5 and i_3. If the sequences are completed, the tree expansion is reverted to its initial state and attached at the respective DC node. This maintains a compact tree during runtime for faster traversal and reflects the non-greedy matching approach. Seq. 1 does not require a tree expansion, since i_1 is *directly* followed by i_2 in order to match successfully.

In summary, we have presented an efficient instruction-screening algorithm integrated as part of QEMU. At the heart of the algorithm we employ a well known string matching technique which was extended such that a variety of accidental and malicious attacks against hardware flaws can be detected. A major challenge was the spatial distribution of instructions inside a given search-pattern. The effective screening of executables during execution becomes possible.

In the next section our experimental evaluation is presented.

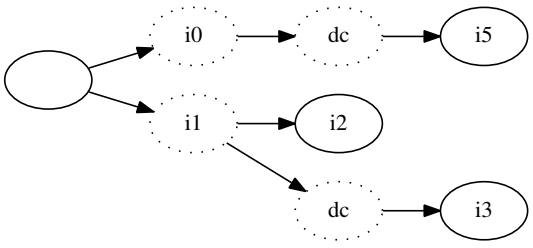

Fig. 5. Computed tree

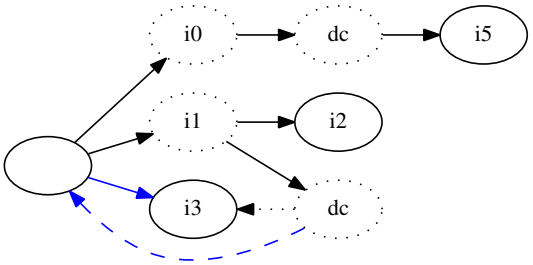

Fig. 6. Temporal expanded tree for i_3

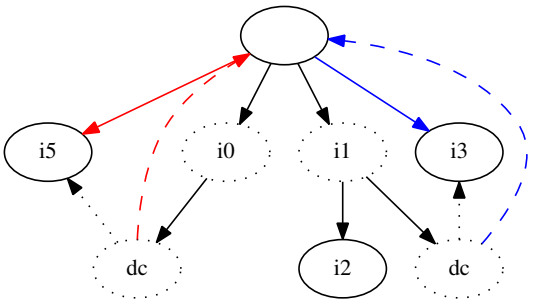

Fig. 7. Temporal expanded tree for i_3 and for i_5

V. Case Studies and Results

For the evaluation of our approach we need a set of benchmarks. However, programs with malicious intents focusing on hardware flaws are typically not widely available. Hence, we integrated specific instruction sequences – implementing different attacks – as a set of characteristic benchmarks. The different scenarios, used to create a set of handcrafted programs, are briefly presented and followed by a detailed discussion of our results.

A. Creation of Benchmarks

A variety of malicious instruction patterns were added to some well known Linux user-land programs. One category of benchmarks includes errata-related bugs. The other includes the Rowhammer-attack and a cryptographic algorithm inside the source code. In the following, we describe the different characteristic flaws which have been considered in our benchmarks.

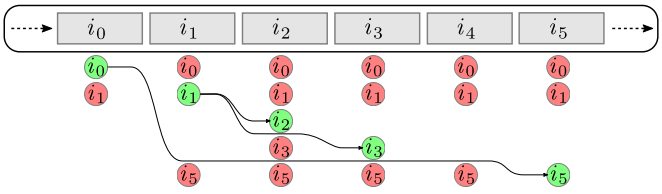

Fig. 8. Simultaneous sequence matching

1) Errata-Instructions: Several silicon bugs have been discovered after the developed and fabricated products have been shipped. In general, until a fix for such issues is available, there is a time window in which systems are susceptible to attacks exploiting these bugs. In the following we give two prominent examples which we also used in our benchmarks:

- The **Cyrix Coma-Bug** [24] is a flaw in the x86 processor-series (6x86, 6x86L, and 6x86MX) manufacturer by Cyrix. When executed, a non-privileged program is able to lock the processor completely.
- The **Pentium f00f-Bug** [25] can transition the affected processors to a state, where only a reset is able to recover the system from this state. The flaw affects the locked compare and exchange instruction of eight bytes in register eax.

2) Active Attack Scenarios: Beside flaws in the fabricated product, there are attacks that exploit vulnerabilities of architectural features or the specific feature size of which components are made. Again, provide two concrete example to be used later:

- The **Rowhammer-effect** is an ongoing threat [7]. The origin of this susceptibility is the decreasing feature size which is used to fabricate *dynamic RAM* (DRAM) chips (even in hand held devices [26]). It is possible to exploit this property with the purpose of data falsification or to escalate privileges (e.g. to gain administrative rights).
- **Ransomware** is an increasing threat. Victims to ransomware-based attacks often are left with encrypted data with the purpose to blackmail the victims to gain back their data. With the availability of dedicated cryptographic instructions (e.g. AES-NI), cryptographic routines are sped up significantly. These patterns contain characteristic instruction sequences which are detectable.

The provided benchmarks are based on well-understood instruction patterns and were chosen for demonstration. Our proposed approach is only applicable if the malicious instruction pattern is known in advance.

B. Evaluation

Basis for the implementation of our approach is the latest stable version of QEMU (2.8.0). The translation inside QEMU has been extended while the internal data structures and other functions have been kept. All programs have been executed in QEMU and invoked by passing a 10 Megabyte file. We tested our approach for different the scenarios, i.e. errata-based, Rowhammer and a cryptographic routine.

Modified versions of well-known Linux user-land programs (*copy*, *diff* and *tar*) have been used as benchmarks, since these could easily be compromised without being noticed. Additionally, *gzip* was augmented with *multiple* bug- or attack-scenarios, such that the benefit of the Aho-Corasick algorithm based screening approach of QEMU could be examined in detail.

Table I summarizes the obtained results. The first column contains the name of the augmented program. Each program was run with different bug- or attack-scenarios, which is triggered by a specific instruction sequence which was included in the respective binary. Each bug- or attack-scenario individually requires specific search patterns. These configurations are indicated by the symbols next to the program name.

TABLE I
RESULTS SUMMARY

Application Details	Execution Details				Found
Name Incl. Bug	Native	QEMU	Extension	Increase	
– – – – – –	[msec]	[off][msec]	[on][msec]	[in %]	
cp †	27	177	193	9.04	✓
cp ‡	29	161	189	17.39	✓
cp ⋆	27	178	202	13.48	✓
cp ∗	28	180	185	2.78	✓
diff †	1	140	164	17.14	✓
diff ‡	2	158	162	2.53	✓
diff ⋆	2	165	168	1.82	✓
diff ∗	1	157	162	3.18	✓
tar †	34	260	271	4.23	✓
tar ‡	32	268	276	2.99	✓
tar ⋆	36	243	265	9.05	✓
tar ∗	35	230	255	10.87	✓
gzip † ‡	138	369	394	6.78	✓
gzip ‡ ⋆	132	388	408	5.15	✓
gzip ⋆ ∗	147	373	407	9.12	✓
gzip † ‡ ⋆	140	394	452	14.72	✓
gzip † ‡ ⋆ ∗	151	367	423	15.26	✓

† Cyrix Coma-Bug included in binary.
‡ Intel Pentium f00f-Bug included in binary.
⋆ Rowhammer (Memory Disturbance) attack included in binary.
∗ Random assembler pattern included in binary.

The second column presents the results of a native execution on the host system. No virtualization (i.e. KVM) or emulation environment was used. This column present the regular execution time (all in milliseconds) as experienced by any user on any system. A compute server, equipped with a Quad-Core Xeon Processor (Intel Xeon E3-1275) running Fedora Linux, was used to run the benchmarks.

In the third column (QEMU) shows the runtime of the respective binaries in emulation mode of QEMU (no KVM support). This execution represents the golden reference for our own implementation. Next, the fourth column contains the results of our extension to QEMU (indicated by "Extension"). During this execution, our methodology was active and screened for the provided search pattern.

Our results are presented in the fifth column. It contains the execution time of the augmented binaries by the provided percentage with respect to the golden reference results (100%). Finally, the last column contains indicates, that each bug- or attack-scenario was successfully detected. It must be noted, that *all* search pattern were active during all of the experiments without yielding neither false-negatives nor false-positives.

C. Observations

The experimental results indicate the successful system protection against the characteristic bug- or attack-scenarios as discussed before. It can be noted, that the increase in runtime during active detection is less than 18% in any given case with respect to the golden reference results. An average increased of runtime of 7.8% was observed. Obviously, there is a static

runtime penalty in comparison to the native execution without the translation layer introduced by QEMU. Interestingly, there is no additional penalty, when multiple search patterns are matched simultaneously by our methodology.

1) Errata-Instructions: Search patterns such as the the coma- and the f00f-bug can be reliably detected. Due to the severity of these bugs, susceptible processors would transition to a non-recoverable state which has to be prevented. Since almost every processor generation will yield a large errata document, this methodology has a possibly large field of application. Third-party legacy software could contain malicious instruction sequences by accident. In these cases our methodology provides system protection while the software *can* still be used. In order to determine the robustness of this extension, we included a random (widely distributed, but unique) search pattern to the binary. The presence of this random patterns was also determined reliably.

2) Active Attack Scenarios: For active attack scenarios we chose the Rowhammer-attack (based on the clflush instruction). The pattern, implementing the Rowhammer-attack, was detected by the extension to QEMU since it can be mapped to a specific instruction sequence. It must be noted, that a Rowhammer-attack can also be induced by behavioral cache-attacks [27] which can be detected, if the instruction sequence is known in advance. Additionally, a standalone program was run, implementing a minimalistic cryptographic AES-function. We extracted the characteristic portion of an AES round as a search pattern and fed it to our extension. The detection of an integrated AES subroutine was reliable in our experiments. From a technical point of view, the detection of AES-NI instructions is even more reliable, since a specific opcode will be present in the instruction-stream which can be detected easily.

This discussion of results can be concluded as follows: The proposed approach has proven to be effective in the domain of runtime instruction screening with a focus on hardware flaws. Our intended use-case for this methodology is one complete execution inside QEMU. After the absence of malicious instruction sequences has been verified, the software can be run natively on the host's hardware. This way, the user can be sure (even in case of legacy or third-party software) that no harmful instruction sequences will compromise the host system. However, from this work the need for a database (such as well known for software-viruses) becomes imperative.

VI. Conclusion and Future Work

We have introduced a framework to detect software-based attacks which exploit hardware flaws. Our approach performs instruction screening during dynamic program execution inside QEMU. The efficiency of our solution is based on an extended Aho-Corasick string matching algorithm which allows for parallel matching of search patterns while maintaining a linear complexity. Our approach brings protection against hardware flaws which reach from simple errata-based flaws in fabricated hardware to feature size related vulnerabilities. Many of these can be fixed by BIOS, microcode or firmware updates, but typically several month pass until they are readily available. An alternative is additional hardware which *can* prevent such attacks [22]. But this hardware is expensive and impracticable after fabrication. In contrast the proposed solution offers the possibility to run software in a safe environment, given that the speed degradation is acceptable and search patterns are provided. As future work we propose screening directly on the KVM layer to make this approach applicable to fully virtualized environments.This way this approach can be transferred to the kernel space, thus establishing an even lower bound of protection and increased execution speed.

References

[1] E. C. R. Council, "The economic impacts of the august 2003 blackout," *Washington, DC*, 2004.

[2] J. Hernandez-Castro, E. Cartwright, and A. Stepanova, "Economic analysis of ransomware," *CoRR*, 2017.

[3] P. Patra, "On the cusp of a validation wall," *Design Test of Computers*, pp. 193–196, 2007.

[4] U. Guin, D. DiMase, and M. Tehranipoor, "Counterfeit integrated circuits: detection, avoidance, and the challenges ahead," *Journal of Electronic Testing*, pp. 9–23, 2014.

[5] A. Baumann, "Hardware is the new software," in *HotOS*, 2017, pp. 132–137.

[6] C. Domas, "Breaking the x86 ISA," *Black Hat*, 2017.

[7] O. Mutlu, "The rowhammer problem and other issues we may face as memory becomes denser," in *Design, Automation and Test in Europe*, 2017, pp. 1116–1121.

[8] N. Karimi, A. K. Kanuparthi, X. Wang, O. Sinanoglu, and R. Karri, "(magic): Malicious aging in circuits/cores," *TACO*, p. 5, 2015.

[9] H. Zhang, L. Bauer, M. A. Kochte, E. Schneider, H. J. Wunderlich, and J. Henkel, "Aging resilience and fault tolerance in runtime reconfigurable architectures," *Trans. on Computers*, pp. 957–970, 2017.

[10] F. Bellard, "QEMU, a fast and portable dynamic translator." in *USENIX ATC*, 2005, pp. 41–46.

[11] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, pp. 333–340, 1975.

[12] P. Jayaraman and R. Parthasarathi, "A survey on post-silicon functional validation for multicore architectures," *ACM Comput. Surv.*, pp. 61:1–61:30, 2017.

[13] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," *Journal in Computer Virology*, pp. 251–266, 2008.

[14] A. A. E. Elhadi, M. A. Maarof, B. I. Barry, and H. Hamza, "Enhancing the detection of metamorphic malware using call graphs," *Computers & Security*, pp. 62–78, 2014.

[15] S. Alam, R. N. Horspool, I. Traore, and I. Sogukpinar, "A framework for metamorphic malware analysis and real-time detection," *Computers & Security*, pp. 212–233, 2015.

[16] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Mascat: Stopping microarchitectural attacks before execution." *IACR*, p. 1196, 2016.

[17] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "Anvil: Software-based protection against next-generation rowhammer attacks," *ACM SIGPLAN Notices*, pp. 743–755, 2016.

[18] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86," in *USENIX ATC*, 2008, pp. 293–306.

[19] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas, "Patching processor design errors with programmable hardware," *Microelectroics Journal*, pp. 12–25, 2007.

[20] S. Narayanasamy, B. Carneal, and B. Calder, "Patching processor design errors," in *ICCD*, 2006, pp. 491–498.

[21] I. Wagner and V. Bertacco, "Caspar: Hardware patching for multicore processors," in *Design, Automation and Test in Europe*, 2009, pp. 658–663.

[22] K. Schmitz, A. Chandrasekharan, J. G. Filho, D. Große, and R. Drechsler, "Trust is good, control is better: Hardware-based instruction-replacement for reliable processor-ips," in *ASP-DAC*, 2017, pp. 57–62.

[23] R. R. S.M. Vidanagamachchi, S.D. Dewasurendra and M.Niranjan, "Commentz-walter: Any better than aho-corasick for peptide identification?" in *Int'l Journal of Research in Comp. Science*, 2012, pp. 33–37.

[24] A. D. Balsa, "The cyrix 6x86 coma bug," https://lkml.org/lkml/1997/11/12/129, 1997.

[25] R. R. Collins, "The intel pentium f00f bug description and workarounds," *Dr. Dobb's Journal*, 1997.

[26] M. S. Inci, T. Eisenbarth, and B. Sunar, "Hit by the bus: Qos degradation attack on android," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 716–727.

[27] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox - practical cache attacks in javascript," *CoRR*, 2015.