Formal Verification of Secure Boot Process

1st Sriram Vasudevan
Temasek Lab @ NTU
Nanyang Technological University
Singapore
sriram.vasudevan@ntu.edu.sg

2nd Prasanna Ravi
Temasek Lab @ NTU
Nanyang Technological University
Singapore
prasanna.ravi@ntu.edu.sg

3rd Arpan Jati
Temasek Lab @ NTU
Nanyang Technological University
Singapore
arpan.jati@ntu.edu.sg

4th Shivam Bhasin
Temasek Lab @ NTU
Nanyang Technological University
Singapore
sbhasin@ntu.edu.sg

5th Anupam Chattopadhyay
School of Computer Science and Engineering
Nanyang Technological University
Singapore
anupam@ntu.edu.sg

***Abstract*—Formal verification is widely used for checking the correctness of a system with respect to the succinct properties at early design phase. In recent times, these methods are also adopted to validate the security of a system by rightly abstracting the security-oriented properties. In this work, we focus on a recently reported attack on the secure boot process of Zynq-7000 platform. We study the vulnerability with formal analysis of its boot loader code. The challenge is to model the system and identify the right set of properties so that, the vulnerability is exposed quickly. It is shown that the UPPAAL model checking analysis can be used, which helps to find the vulnerability instantaneously with the help of four properties and a virtual memory usage peak of about 50MB to test each property. To the best of our knowledge, we perform the first analysis of UPPAAL on a concrete software implementation and perform a case study on a real security vulnerability reported in a CVE [1].**

***Index Terms*—formal verification, embedded system, vulnerability analysis, query.**

I. Introduction

The importance of security, particularly for critical cyber-physical systems cannot be overstated. As device complexity increases and the world increasingly becomes digitized, it is even more important that the systems designed to handle these capabilities are secure and the infrastructure is protected. As much as there are external security mechanisms that protect them, it is also important that the systems themselves inherently do not have vulnerabilities [2]. A security break in these infrastructures is always devastating and could cause a huge loss. For instance the cyber-attacks notpetya and wannacry has been estimated to have caused billions of dollars loss worldwide [3]. On the other hand, it could expose sensitive and personal data of citizens and cause harm to them. One such example is the Zoom application that suffered a data breach in 2020, resulted in the personal data of around 500 million users getting exposed [4].

To make sure the infrastructure deployed does not suffer from such vulnerabilities, developers have to build systems that are fool-proof. However, there are always vulnerabilities that are found, reported and corrected which are known as Common Vulnerabilities and Exposures (CVE) [5]. In this work, we focus on one such CVE that reported a security flaw in the First Stage Boot Loader (FSBL) used in the Secure Boot procedure of Zynq 7000 SoC [1] that allows to bypass RSA authentication, and mount an unauthenticated application on the target device [6]. The main contribution of our work is to demonstrate the ability of formal verification tools such as UPPAAL to model security software and show successful identification of the security vulnerability. Thus, our work shows that this particular CVE at least could have been easily avoided through utilization of formal verification based tools. Moreover, we demonstrate how security properties can be modelled as formal properties within the UPPAAL system, aiding in successful identification of security vulnerabilities. To the best of our knowledge, this the first work to study a concrete security software implementation using UPPAAL to demonstrate its ability to identify concrete security flaws. Our work therefore stresses the need towards the use of mandatory and rigorous formal verification of security software, particularly deployed in embedded devices.

Prior Work

There have been multiple contributions in the literature that use formal verification, specifically model checking to verify system designs [7] [8] [9]. The work by Avalle et al. [7] provides a survey on the list of works that have been done to automatically obtain formal security proofs on models resembling the source code. It provides details on the existing works done using different abstract model languages. The work by Hofer et al. [8] focuses on a survey on the application of formal verification to improve security in IoT devices. The work by Kulik et al. [9] is a more recent survey on the practical formal methods for security. This work provides a detailed account of all formal verification methods, the approaches

This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research & Development Programme (Cyber-Hardware Forensic & Assurance Evaluation R&D Programme <NRF2018NCRNCR009-0001>). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the view of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

979-8-3503-4859-0/DATE24/© 2024 EDAA

and their application in the digital domain. Recently, Moller et al. [10] formally verify a small set of security properties using specification of the OpenTitan boot code, using model checking tools such as UPPAAL and CBMC. However, they carry out their analysis using the high-level specification of the OpenTitan code, and were unable to identify any new vulnerability. Moreover, they speculate whether their analysis using UPPAAL can scale to the actual software implementation of the secure boot code. In this work, we perform the first analysis with UPPAAL on a concrete software implementation and perform a case study on a real security vulnerability reported in a CVE [1].

II. Preliminaries

In this sub-section, we provide a background on the UPPAAL and CBMC tools.

A. UPPAAL

UPPAAL is a formal verification environment that is used to model, validate and verify real-time embedded systems, which can be defined as networks of several automata [11]. It allows system designers to abstract the logic of a system as a set of states/nodes and multiple transitions between them. Once the logic abstraction is done, the environment allows to simulate it based on guard conditions and timing delays. Every simulation generates a trace that is stored for verification, which is performed using queries for individual states [11]. For our work, we utilized the version 4.1.26-1 of the UPPAAL tool, that was hosted on a Windows environment.

B. C-based Model Checking (CBMC)

CBMC is a formal verification tool that uses a bounded model checker to verify C and C++ programs [12]. This tool supports multiple compiler extensions that are provided by visual studio and gcc. The main purpose of this tool is to verify the memory safety that includes the bounds for pointers, array and user-defined assertions. This tool works by unwinding the loops in the programs and passing the equations to a decision maker. The version of the tool installed is 5.12 on an Ubuntu environment.

III. Security Bug in Zynq-7000 Secure Boot Software

The central component of a secure boot sequence is the *secure boot image*, stored within a Non-Volatile Memory (NVM) such as SD Card, NOR Flash or NAND Flash accessible to the CPU. It consists of various partitions that will be sequentially loaded in a secure manner after authentication and decryption into the appropriate locations within the Zynq-7000 SoC [13]. The important components of the boot image are as follows:

1) Boot Image header (BIH)
2) Partition Header Table (PHT)
3) First Stage Boot Loader (FSBL)
4) One or more Programmable Logic (PL) partitions
5) One or more Programmable System (PS) partitions

Except for the Boot Image Header (BIH), all the above listed partitions can be authenticated using the RSA-2048 digital signature present in the boot image. Upon Power On Reset (POR), the processor executes the BootROM code, which first reads the BIH to locate the First Stage Boot Loader (FSBL) within the boot image. The BootROM then authenticates the FSBL and upon successful authentication, decrypts the FSBL into secure On-Chip Memory (OCM) and then passes control to the FSBL. The FSBL is then used to secure load (authenticate and/or decrypt) all the executable partitions for the processor as well as the FSBL in the boot image. The first significant operation performed by the FSBL during an authenticated secure boot is to retrieve the Partition Header Table (PHT) from the boot image and authenticate the PHT.

The PHT is a critical component which contains metadata information about every other executable PL/PS partition in the boot image. This includes encrypted partition size, decrypted partition size, total partition size including the RSA signature, location of the partition, encryption and authentication status of the partition among others. If an attacker is able to bypass PHT authentication, he/she can mount a tampered PHT that can be used to execute an unauthenticated application on the Zynq device. Thus, it is critical to authenticate the PHT data. Thus, the FSBL starts with authenticating the PHT, and if successful, the FSBL starts loading the different PS and PL partitions based on information present in the PHT. Upon failure, the FSBL simply aborts the secure boot procedure. Unlike the BootROM code, which cannot be accessed or changed, the FSBL code is open-source and therefore can be easily analyzed for vulnerabilities [14].

Algorithm 1: FSBL Implementation in Zynq-7000

```
Result: Partition Move
InitFSBL();
NVMRetrieve PartitionHeader_1 from NVM;
if RSA Authentication is Necessary then
    NVMRetrieve PartitionHeader_2, PHTCertificate
     from NVM;
    Authenticate Tmp_Variable, PHTCertificate;
    if Authentication Fails then
        FsblFallback();
    else
        Proceed();
    end
else
    Proceed();
end
for All Partitions in PHT do
    CurrentPartitionMetaData =
     Extract(PartitionHeader_1, PartitionCount);
    Authenticate and Decrypt CurrentPartitionData if
     required;
    Load CurrentPartition based on
     CurrentPartitionMetaData;
end
```

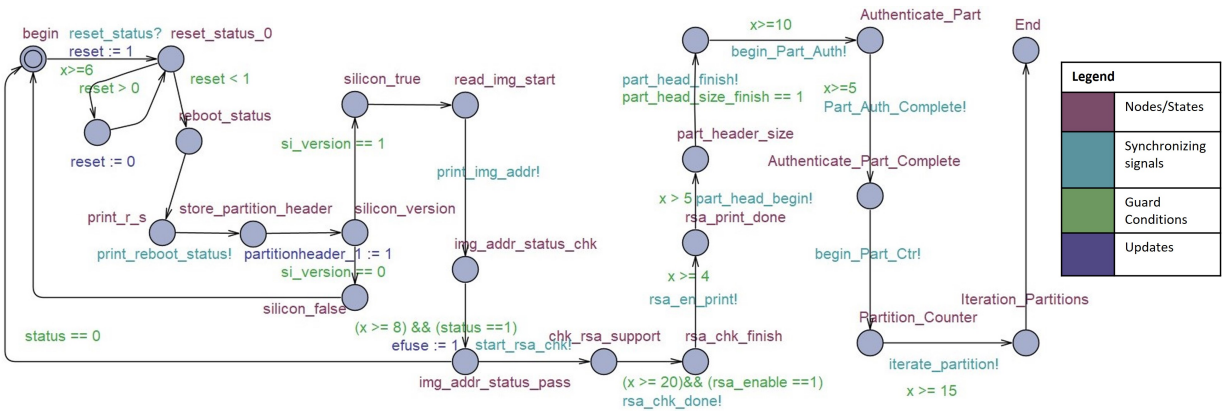

Fig. 1. UPPAAL Model - Zynq 7000 Img Mover System

A. CVE 2022-23822 to Bypass Authentication of PHT in FSBL

In this work, we specifically focus on the authentication procedure of PHT, which contains the security flaw that allows to bypass its authentication, reported in the FSBL software [1] [6]. The logic is implemented within the image_mover.c source file in the *embeddedsw* project which can be found in [14]. We refer to Algorithm 1 for a pseudo-code of the same procedure, and the outline of the PHT authentication procedure is as follows:

1) The FSBL retrieves the PHT data from NVM and stores it in a variable denoted as PartitionHeader_1 highlighted in red in Alg.1.
2) The FSBL then checks whether RSA authentication is mandatory using the RSA eFUSE value. If enabled, the FSBL again queries the NVM to retrieve the PHT (PartitionHeader_2) along with its authentication certificate, denoted as PHTCertificate. The data in PartitionHeader_2 is authenticated using the PHTCertificate and if the verification passes, the FSBL then uses the PHT data in PartitionHeader_1 to load the other PS/PL partitions in the boot image.

In other words, FSBL authenticates PHT data in PartitionHeader_2, but uses the unauthenticated data in PartitionHeader_1 for loading the PS/PL partitions. Since the PHT data is fetched from the NVM (external memory), it can therefore be actively tampered by an attacker.

We realize a practical attack using an in-house built SD card multiplexer which can switch between two SD cards to communicate with the Zynq device. The first SD card (SD Card 1) contains a boot image with a tampered PHT corresponding to an unauthenticated software PS partition, while the second SD card (SD Card 2) contains a boot image with a valid PHT corresponding to authenticated partitions. But, the SD card 2 contains the unauthenticated software PS partition that the attacker intends to load onto the target device. Please refer to Fig.2 for the attack setup. The attack is carried out in the following manner:

1) We first boot the Zynq device with SD card 1. Once the FSBL retrieves the tampered PHT PartitionHeader_1 from SD card 1, we manually switch from SD card 1 to SD card 2. For our demonstration, we add a suitable delay in the FSBL to enable the manual switch. However, this can be automated as the timing of switch is constant for the Zynq device upon power up.
2) The FSBL now checks for the RSA eFuse, and if enabled, retrieves the PHT again (PartitionHeader_2) along with its signature. But, since the Zynq device is connected to SD card 2 with a valid PHT, the PHT should be authenticated successfully by the FSBL. The flaw in the FSBL software ensures that the tampered PHT PartitionHeader_1 is used for secure boot. This should ensure that the attack application is mounted in an unauthenticated manner on the Zynq device.

We were able to experimentally verify that our attack setup is able to successfully exploit the security vulnerability, and mount an unauthenticated application on the target device.

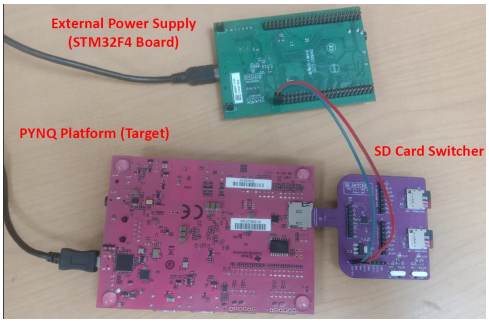

Fig. 2. Attack Setup for PHT Authentication Bypass using SD Card Switcher

IV. UPPAAL Modeling

Figure 1 represents the UPPAAL model, which is an abstraction of the logic corresponding to the implementation of FSBL. This system contains basic components which are termed as templates, which communicate with each other and with the system through synchronizing signals. In order to study the CVE reported in Zynq 7000 FSBL, it was chosen to model only those specific functions and logic that are of interest for replicating and understanding the CVE [1].

A. Templates modelled in the UPPAAL system

The system (image_mover) as shown in Figure 1 starts at the node *begin*. In the following, we explain the critical operations within the FSBL software relevant to the security flaw.

Reset: In order for this program (image_mover.c) to execute, the first requirement is that of a system reset, modeled as a separate template named as *Reset*.

Retrieval of PartitionHeader: The partition header is designed as a separate template Partition_head. The partition header that is retrieved the first time is stored in a variable partitionheader_1 and is assumed to have an input value of integer 1 (assumption to be a simple integer input for the purpose of modelling).

The partition header is read again the second time (based on the FSBL implementation) and stored as partitionheader_2 and is assumed to have an integer input value equal to 1 (since it is the same partitionheader that is being read again).

Checking RSA Authentication Enable Status in eFuse: The process to check the value of the RSA Enable Status in the eFuse to ensure mandatory RSA authentication of the boot image is captured in a separate template rsa_chk. The start_rsa_chk signal is used to synchronize with the image_mover system. The value of efuse_RSA to be 1 is checked to pass the RSA check. An rsa_chk_done signal is used for synchronization.

PHT Authentication Operation: The next is to authenticate the partition header, designed as a template Partition_Auth, shown in Figure 3. A signal begin_Part_Auth is used to synchronize with the image_mover system. For the authentication to be successful, the status_auth is checked as a guard condition to be equal to XST_SUCCESS (a Zynq 7000 board constant to denote a successful event). If it is XST_FAILURE (a Zynq 7000 board constant to denote a failure event), then the FSBL fallback logic is executed.

Iterating over PL/PS Partitions: After the authentication of the partition is complete and successful, the next step is to begin a partition counter and use this to iterate the number of partitions as shown in Figure 4. This is a separate template, Part_Counter. Here, the counter variable partition_counter is initialized to the number of partitions in the image. The partition header (PartitionHeader_1) obtained before is used and each partition is extracted and the counter is incremented on each iteration. After all the partitions are extracted, this template ends. A signal iterate_partition is used for synchronization after which the system ends, which indicates the end of the FSBL operation.

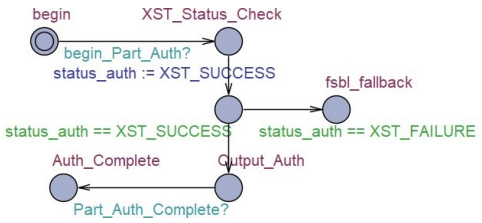

Fig. 3. UPPAAL Model - Partition Authentication Template

B. Formal Verification of FSBL for Presence of CVE using UPPAAL

The properties that can be tested on the UPPAAL framework are deadlock, reachability, safety and liveliness. In the following, we show that the security properties to be satisfied can be adapted into the aforementioned properties that can be tested for satisfiability using the UPPAAL framework.

In order to verify the satisfiability of these properties, the idea is to build queries to verify each one of the properties described above. UPPAAL specifically uses a very simple version of Computation Tree Logic and the query language uses two types of formulae (i) Node Formulae that describe the individual nodes/states and (ii) Path Formulae that describe paths/traces in the model that are being analyzed [15]. We now describe each of these four properties and how we are able to test for the security flaw using the same.

a) Deadlock: This property denotes that there are no outgoing transitions from any particular node in the design. UPPAAL verifier can detect the presence of any deadlock in the model that is built using the query "A[] not deadlock". The verification result of this test is "property is satisfied", which implies no deadlock in the current design. The resident and virtual memory usage peaks for this test was 12.8MB and 51.2MB respectively. Verification of the deadlock property cannot be used to indicate any security flaw, as it only checks for proper functioning of the logic without any deadlock.

b) Safety: This property verifies a situation where a terrible incident would never occur. For instance, in the operation of a power generator with a backup, there would never be a blackout of power in the distribution channel. These properties are expressed as "A[]" in the verifier. Table I lists the queries built specifically to test the safety properties of all the nodes in the design. For each node in the system, the guard condition of that node or a related node is used to build a specific query to check if it satisfies that safety property. Each query that is built can satisfy or not satisfy the condition that they are built for which are indicated as expected result and the actual result of executing the query. For example, query 2 in Table I, the condition being checked is to see if the node silicon_version is reached safely. This occurs with the condition that the value of si_version is equal to 1 and the expected result of executing the query is that it is satisfied. Any other value of si_version implies that the condition will not be satisfied (which is expected as shown in query 3). Thus, even if the safety property is not satisfied, if the result of a query is as expected, then the system still remains safe.

Thus, it is important to note that there are certain queries such as query 3, 5, 7, 10, 12 and 13, whose expected result is failure to satisfy and we also observe the same from our experiments. However, examining each of these queries separately indicate to us that the failure to satisfy these properties do not result in security vulnerability or flaw. For instance, unsatisfiability of property in query 3 merely denote that the device somehow received the wrong silicon version, and it would only simply result in failure of secure boot, and thus cannot be exploited by an attacker. The same reasoning can also be given for

other queries 5, 7, 10, 12 and 13 whose unsatisfiability can be justified, and does not indicate any security weakness, but merely corrupt the functioning of the device.

However, a specific query that is of interest to us is the query 14 in Table I, which checks whether the unsafe condition of reaching the end of FSBL operation occurs, even if partitionheader_1 is not equal to partitionheader_2. This property therefore checks whether partition header obtained as input to the functions at multiple instances are the same. In any ideal situation, this property should always be satisfied as the two instances of the partition header represent the same input. However, since these two are external inputs to the system there is always a chance of the input being changed by means of an attack. Hence, while executing this query, we modify the input to the partitionheader_2 from the integer value 1 to 0 and this causes the query to fail for the given input of the partitionheader. Thus, the safety property of the system for this query fails.

If the end of FSBL operation is reached, that means that it is possible to operate over a tampered partitionheader_1 for loading the PS/PL partitions even if partitionheader_2 is authenticated, which clearly indicates the security flaw.

c) Liveliness: This property verifies if a particular scenario would eventually become true at some point during the working of the design. A simple example would be a light bulb circuit verification, where the light bulb should turn ON once the switch is turned ON. These properties are expressed as "A<>".

A list of queries similar to the safety properties are constructed for verifying the liveliness as shown in Table II. All the queries that were checked for safety is replicated for liveliness property. The particular query of interest for this design is the Serial Number 9 in the Table II, which tests whether the FSBL operation is complete even if partitionheader_1 is not equal to partitionheader_2. We observe that the liveliness property is also satisfied, even if partitionheader_1 is not equal to partitionheader_2. This indicates that it is possible that data in partitionheader_1 which is used to load the other PS/PL partitions can be tampered by the attacker, which also clearly indicates a security flaw.

d) Reachability: Given a node/state and given a formula, the verifier checks if it can be satisfied by any of the reachable nodes. An example of this property is a client server model where a message sent by the client to server can be verified if it can reach the server using this property. The reachability property is represented as "E<>" along with the name of the node for which reachability is to be tested. For instance, to check the reachability of the node reboot_status, the syntax would be $E <> \ Img\_Mover.reboot\_status$. For the design in Section IV, the reachability property is tested for every node in the image_mover system. This property is satisfied for all the nodes in the design. This indicates that every node in the design is reachable even if partitionheader_1 is not equal to partitionheader_2, which is as expected because there is no condition to check if the values of the partitionheader remains the same which also clearly indicates a flaw in the design.

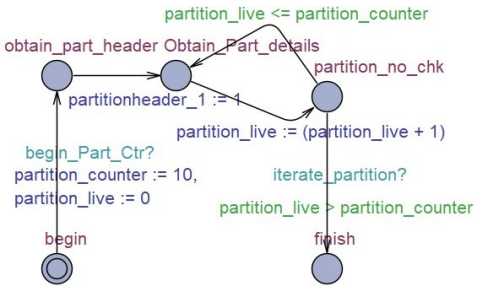

Fig. 4. UPPAAL Model - Partition Counter Template

By studying these four properties, it is evident that it is possible to reach system states when the partitionheader_1 and partitionheader_2 data are not the same, and since partitionheader_1 is used to load the PS/PL partitions, it clearly identifies the flaw reported in the CVE [5]. Thus, formal verification of the design logic by understanding the implementation has helped to identify the CVE. On the other hand, building a model from system specification and verifying it before implementation could have avoided a critical CVE in the design of the Zynq 7000 FSBL.

The verification time for each query was in the order of milliseconds (for instance, the deadlock property took 7ms to be verified while the rest of the properties each took a time less than this). The resident memory usage and the virtual memory usage peaks was on an average 12MB and 51MB respectively for each query listed in the Tables I and II.

TABLE I
UPPAAL QUERIES- SAFETY

Sl	Queries	Expect Result	Res
1	A[] Img_Mover.reboot_status imply reset<=0	Yes	Yes
2	A[] Img_Mover.silicon_version imply si_version == 1	Yes	Yes
3	A[] Img_Mover.silicon_version imply si_version == 0	No	No
4	A[] Img_Mover.silicon_true imply si_version == 1	Yes	Yes
5	A[] Img_Mover.silicon_true imply si_version >1	No	No
6	A[] Img_Mover.img_addr_status_chk imply img_addr == 1	Yes	Yes
7	A[] Img_Mover.img_addr_status_chk imply img_addr == 0	No	No
8	A[] Img_Mover.rsa_chk_finish imply (Img_Mover.x >= 20 && rsa_enable ==1)	Yes	Yes
9	A[] Img_Mover.Authenticate_Part_Complete imply (status_auth== 1)	Yes	Yes
10	A[] Img_Mover.Authenticate_Part_Complete imply (status_auth== 0)	No	No
11	A[] Img_Mover.Iteration_Partitions imply (Img_Mover.x >= 15)	Yes	Yes
12	A[] Img_Mover.Iteration_Partitions imply (Img_Mover.x == 15)	No	No
13	A[] Img_Mover.Iteration_Partitions imply (Img_Mover.x <= 15)	No	No
14	A[] Img_Mover.End imply (partitionheader_1 == partitionheader_2)	Yes	No

V. CBMC Modeling

In order to further test the Zynq 7000 FSBL c-code, a bounded model checker CBMC is used to supplement the results of the UPPAAL model checker. The logic has been studied using the UPPAAL tool and the code is now verified using CBMC.

a) CBMC Setup: The CBMC tool is installed in a virtual box that hosts an Ubuntu operating system.

b) Testing the FSBL code: The entire FSBL code is saved in a working folder along with its header files. There are two tests that are performed to verify the FSBL code. First is testing the main.c function and second is the c file under study here, which is image_mover.c. For the main.c function, the following tests are performed which are, cbmc main.c –show-claims, cbmc main.c –show-vcc -unwind 1 and cbmc main.c –function main –unwind 1.

The reason for testing the main.c with an unwind parameter is the presence of a while loop. If the CBMC is executed on this main program without an unwind parameter, it is noted that the unwinding never stops. The possible reason is that the in-built simplifier of the CBMC is unable to determine a run-time loop bound. However, with the unwind parameter, the loop ends and the verification of the main.c is successful.

Second is the testing of the image_mover.c program file itself. This program file does not have a main function and hence there is no entry point for this c program. Therefore, every function is independently verified. The following are the tests performed. cbmc image_mover.c –show-claims, cbmc image_mover.c –show-vcc and cbmc image_mover.c –function **function_name**.

The function_name is replaced with the function that is being tested. Every function in this c program is tested and the verification result is successful. The CBMC testing of the FSBL code has aided in verifying the logic at code level. However, it has not assisted in finding the logical flaw reported in the Section III. Thus, we can clearly observe the superior ability of UPPAAL in identifying the security vulnerability compared to CBMC. While we have only performed the direct comparison for this particular vulnerability, we leave the concrete comparison of the same for several types of vulnerabilities for future work.

VI. Conclusion

In this work, we have presented a CVE that was discovered in the FSBL of Zynq 7000 that could potentially lead to the entire hardware being taken over by attackers. Given that we are already aware of this vulnerability, in this work we have explored the possible options to detect and avoid similar vulnerabilities in future FSBL designs.

To demonstrate a methodology to detect these vulnerabilities, we have selected formal verification through modeling. Specifically, we have chosen UPPAAL and CBMC as two methods to formally verify this FSBL design. UPPAAL provides a logic abstraction of the design and CBMC provides a code level analysis of the design. The UPPAAL formal verification shows that the CVE could have been avoided if the system specification was first modelled and verified before its implementation. This methodology is simple, quick and memory conservative. Thus a combination of UPPAAL and CBMC would be an effective methodology to detect vulnerabilities in FSBL flow (logical and at code level).

TABLE II
UPPAAL Queries-Liveliness

SNo	Queries	Expect Result	Res
1	A<>Img_Mover.silicon_version imply si_version == 1	Yes	Yes
2	A<>Img_Mover.reboot_status imply reset <1	Yes	Yes
3	A<>Img_Mover.img_addr_status_chk imply si_version == 1	Yes	Yes
4	A<>Img_Mover.silicon_true imply si_version == 1	Yes	Yes
5	A<>Img_Mover.img_addr_status_chk imply img_addr == 1	Yes	Yes
6	A<>Img_Mover.rsa_chk_finish imply (Img_Mover.x >= 20 && rsa_enable ==1)	Yes	Yes
7	A<>Img_Mover.Authenticate_Part_Complete imply (status_auth == XST_SUCCESS)	Yes	Yes
8	A<>Img_Mover.Iteration_Partitions imply (Img_Mover.x >= 15)	Yes	Yes
9	A<>Img_Mover.End imply (partitionheader_1 == partitionheader_2)	Yes	Yes

References

[1] Xilinx Zynq FSBL CVE-2022-23822. https://nvd.nist.gov/vuln/detail/CVE-2022-23822.

[2] S. Parameswaran and T. Wolf, "Embedded systems security—an overview," *Design Automation for Embedded Systems*, vol. 12, pp. 173–183, 2008.

[3] A. Bendiek and M. Schulze, "Attribution: A major challenge for EU cyber sanctions. An analysis of WannaCry, NotPetya, Cloud Hopper, Bundestag Hack and the attack on the OPCW," SWP Research Paper, Tech. Rep., 2021.

[4] D. Kagan, G. F. Alpert, and M. Fire, "Zooming into video conferencing privacy and security threats," *arXiv preprint arXiv:2007.01059*, 2020.

[5] Common Vulnerabilities Exposures Database. https://www.cvedetails.com/.

[6] P. Ravi, A. Jati, and S. Bhasin, "Breaking RSA Authentication on Zynq-7000 SoC and Beyond: Identification of Critical Security Flaw in FSBL Software," Cryptology ePrint Archive, Paper 2023/1913, 2023. [Online]. Available: https://eprint.iacr.org/2023/1913

[7] M. Avalle, A. Pironti, and R. Sisto, "Formal verification of security protocol implementations: a survey," *Formal Aspects of Computing*, vol. 26, pp. 99–123, 2014.

[8] K. Hofer-Schmitz and B. Stojanović, "Towards formal verification of IoT protocols: A Review," *Computer Networks*, vol. 174, p. 107233, 2020.

[9] T. Kulik, B. Dongol, P. G. Larsen, H. D. Macedo, S. Schneider, P. W. Tran-Jørgensen, and J. Woodcock, "A survey of practical formal methods for security," *Formal Aspects of Computing*, vol. 34, no. 1, pp. 1–39, 2022.

[10] B. Møller, M. Pedersen, and T. Bøgedal, "Formally Verifying Security Properties for OpenTitan Boot Code with Uppaal," *To Appear. MA thesis. AAU*, 2021.

[11] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," *Formal methods for the design of real-time systems*, pp. 200–236, 2004.

[12] E. Clarke and D. Kroening, "ANSI-C bounded model checker user manual," *School of Computer Science, Carnegie Mellon University*, 2006.

[13] Secure Boot of Zynq-7000 SoC. https://docs.xilinx.com/v/u /Qp-MOQc8wC_WNvJcxD9qBIw.

[14] Xilinx Embedded Software (Zynq FSBL). https://github.com/Xilinx/embeddedsw.

[15] Using Queries. https://docs.uppaal.org/gui-reference/yggdrasil/tutorial/using-queries/.