Intermittent bugs in batterielosen IoT Geräten

MARIUS BENDIXEN, Technische Universität Dortmund, Deutschland

ACM Reference Format:
Marius Bendixen. 2021. Intermittent bugs in batterielosen IoT Geräten. 1, 1 (July 2021), 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 EINLEITUNG

Batterie lose IoT Geräte sind Abhängig von direkter Energie, da ihnen die Möglichkeit fehlt diese ausreichend zu speichern. Somit ist eine durchgängige Ausführung des Codes unwahrscheinlich, wodurch häufige Unterbrechungen einkalkuliert werden müssen. Um die Berechnungen ausführen zu können, gibt es unterschiedliche Ansätze. Maeng et al. [2] verwendet Tasks um den Code in Blöcke aufzuteilen, welche dann entweder komplett beendet werden oder nach einem Stromausfall aus dem Urzustand heraus neu berechnet werden müssen. Checkpoints sind den Tasks sehr ähnlich, jedoch wird an diesen der Zustand der energieabhängigen Komponenten in energieunabhänge Speichermedien übertragen, wodurch nach einem Stromausfall das Programm an diesem die Berechnung neu aufnehmen kann.
Checkpoints sind jedoch eine Quelle für neue Herausforderungen in der Programmierung von batterielosen IoT Geräten. Durch die Speicherung an den Checkpoints kann das Programm in Zustände geraten, welche bei einer kontinuierlichen Ausführung nicht erreichbar wären. Maioli et al. [4] hat angesichts dieses Problems ein analytisches Tool mit dem Namen ScEpTIC entwickelt, welches Programme auf Inkonsistenzen untersucht. Besonders werden Speicherinkonsistenzen betrachtet, welche voran gegangene Arbeiten übersehen haben.

2 INTERMITTENT BUGS

Bei der Verwendung eines Checkpointsystemes kann nach einer Unterbrechung des Programms, dieses in einen Zustand übergehen, welcher bei einer kontinuierlichen Ausführung nicht erreichbar wäre. Solch ein Zustandswechsel wird als Intermittent Bug bezeichnet. Die Folgen aus solch einem Bug sind unterschiedlich und reichen von falschen Daten, über falsche Programmabläufe, bis hin zu Speicherlecks und Programmabstürzen.
Maioli et al. [4] beschreibt drei unterschiedliche Bugs, welche bei der Ausführung mit Unterbrechungen auftreten können. Dazu zählt der Data Access Bug, Activation Record Bug und Memory Map Bug. Die meisten Ansätze, welche Bugs in Programmen mit Unterbrechungen untersuchten, befassen sich laut Maioli nur mit dem Data Access Bug, wodurch daraus entwickelte Tools Bugs übersehen und nicht eine sichere Ausführung gewährleisten können. Nur Van Der Woude et al. [7] benennt den Activation Record Bug in seiner Arbeit, jedoch nur als Spezialfall bei Push und Pop Befehl bei der Bearbeitung von Interrupts. Diese werden gelöst, indem alle Pop Befehle als kritisch angesehen werden, da Interrupts nicht vorhersehbar sind.
Der Memory Map Bug wurde, gegen die Behauptung von Maioli et al. [4] jedoch bereits in einer Arbeit erwähnt. Surbatovich et al. [5] benennt bei der Entwicklung eines Debugging Tools, welches ebenso wie ScEpTIC Programme mit Unterbrechungen untersucht, den Memory Map Bug als Spezialfall. Um Probleme in der Ausführung zu erkennen werden bestimmte Muster verwendet. Eines dieser Muster bezieht sich auf die Verwendung von Pointern und die Konsequenzen bei einer Programmunterbrechung. Der daraus resultierende Bug ist der später beschriebene Memory Map Bug.

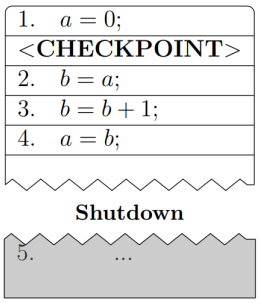

Fig. 1. Beispiel für einen Data Access Bug

2.1 Data Access Bug

Beim Data Access Bug kommt es zu einer Differenz zwischen der kontinuierlichen und unterbrochenen Ausführung, da Werte im NVM (non volotile memory) durch die erneute Ausführung nach einem Stromausfall, doppelt verändert werden.
Fig. 1 zeigt einen einfachen Data Access Bug. Bei dem Ausführen des Codes wird nach dem setzen des Checkpoints der Wert von b inkrementiert und überschreibt den Wert von a. Das Programm bricht ab und beginnt beim der Wiederaufnahme des Programms in Zeile 2. Da der Wert von a bereits einmal inkrementiert wurde und nun noch einmal erhöht wird, existiert eine Differenz zwischen der kontinuierlichen und der unterbrochenen Ausführung.
Maioli et al. [4] fasst den Bug wie folgt zusammen:
Ein Data Access Bug kommt immer dann zu Stande, wenn x eine Adresse im NVM und I_1, ..., I_n eine Folge aus Anweisungen in Maschinensprache ist, so dass

- I_1 lädt einen Wert von der Adresse x
- I_n verändert den Wert und speichert diesen auf Adresse x
- zwischen I_1 und I_n existiert kein Checkpoint

Damit es zu keinem Bug kommt, muss ein Checkpoint zwischen das Laden des Wertes und das Speichern von diesem gesetzt werden.

Author's address: Marius Bendixen, marius.bendixen@udo.edu, Technische Universität Dortmund, Flavusstraße 3, Dortmund, Deutschland.

XXXX-XXXX/2021/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

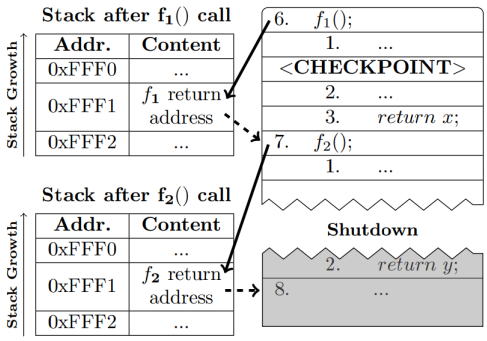

Fig. 2. Beispiel für einen Activation Record Bug

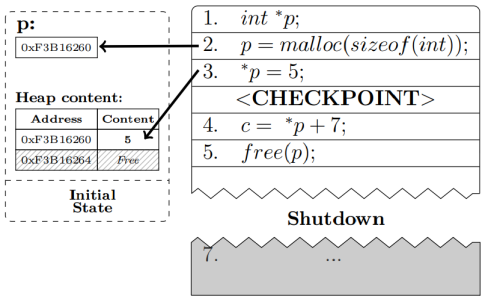

Fig. 3. Beispiel für einen Memory Map Bug

Wichtig ist, dass nicht alle write/read Folgen zu diesem Bug führen, sondern nur bei Daten, welche aus dem NVM geladen und verwendet werden. Dies ist wichtig, damit nicht zu viele Checkpoints gesetzt werden [7].

2.2 Activation Record Bug

Beim Activation Record Bug ist die Ausführungsreihenfolge zwischen der kontinuierlichen und unterbrochenen Ausführung unterschiedlich, da es zu ungewollten Sprüngen und somit falschen Ergebnissen kommen kann. Der Bug tritt normalerweise auf, wenn die return-Adresse einer Funktion durch die einer zukünftigen Funktion überschrieben wird. Das kann in ganz normalen Methoden passieren, jedoch auch in selbst geschrieben Interrupt handlern, welche asynchron zur Ausführung ausgeführt werden [4].
Fig. 2 zeigt einen möglichen Activation Record Bug. Der Checkpoint liegt innerhalb der Funktion $f_1()$ und der Abbruch des Programms geschieht in Funktion $f_2()$. Da durch den Aufruf von Funktion $f_2()$ die return Adresse im Stack verändert wurde und dieser nicht vom Strom abhängig ist, ist der Stack nach des Programmabbruch fehlerhaft und beinhaltet nun eine falsche return Adresse aus sieht von Funktion $f_1()$. Van Der Woude et al. [7] hat auch erkannt, dass Interrupts zu diesem Bug führen. Der von ihm entwickelte Compiler tauscht deswegen alle pop-Befehle mit einem Befehlssatz, welcher die Verteilung der Checkpoints übernimmt. Hier muss auch immer ein Checkpoint gesetzt werden, da die Verletzung der Idempotenz implizit ist [6]. Interrupts tauchen in der Ausführung des Programms nicht konsistent auf, sondern immer unterschiedlich, deswegen können diese im statischen Code nicht unterschieden werden.
Maioli et al. [4] fasst auch den Activation Record Bug wieder wie folgt zusammen:
Ein Activation Record Bug liegt dann vor, wenn der Stack im NVM liegt und eine geordnete Sequenz aus Anweisungen $I_1, ..., I_n$ existiert, so dass:

- I_1 ein call-Befehl für die Funktion f_x ist
- f_x beinhaltet in seiner Ausführung mindestens einen Checkpoint
- I_n ist ein weiterer call-Befehl
- zwischen I_1 und I_n existiert kein Checkpoint

Wichtig ist, dass die Sequenz $I_1, ..., I_n$ nicht den Befehlssatz der Funktion f_x beinhaltet.
Damit kein Bug entsteht, muss ein Checkpoint zwischen die call-Befehle gesetzt werden.

2.3 Memory Map Bug

Ein Memory Map Bug entsteht durch dynamische Speicher-Operationen, wodurch der Zustand des Heaps nicht im Vorhinein bestimmt werden kann. Somit kann es passieren, dass nach einem Stromausfall auf einen Wert zugegriffen werden soll, der jedoch mittlerweile nicht mehr existiert oder einen falschen Wert enthält.
Fig. 3 zeigt einen Memory Map Bug. Vor dem Checkpoint wird durch den malloc Befehl in Zeile 2 der Speicherplatz für die Variable p reserviert. Nach dem Checkpoint bricht nun das Programm ab und nimmt die Ausführung des Programms am Checkpoint wieder auf. Wenn in Zeile 4 mit der Variable p gerechnet wird, ist unklar wie sich das Programm verhalten wird, da die Variable keinen reservierten Speicherplatz mehr besitzt.
Maioli et al. [4] fasst diesen Bug wie folgt zusammen:
Ein Memory Map Bug existiert genau dann, wenn der Heap im NVM liegt und es eine geordnete Sequenz aus Befehlen $I_1, ..., I_n$ gibt, so dass:

- I_1 ist ein load oder store Befehl der den Heap Block x betrifft
- I_n ist ein free oder realloc Befehl der den Heap Block x verändert
- zwischen I_1 und I_n existiert kein Checkpoint.

Um den Bug zu vermeiden, müssen Checkpoints so gesetzt werden, dass die erneute Ausführung von Pointer, dessen Wert nicht mehr sicher ist, vermieden wird. Wichtig ist, dass diese Art von Bugs nicht die Korrektheit des Programms beeinflussen muss. Es kann auch sein, dass die Neuausführung zu Speicherlecks führt, die Ausführung jedoch trotzdem korrekt ist.

3 SCEPTIC

Maioli et al. [4] entwickelte ScEpTIC als Debugging-Tool für Intermittent Execution. Es soll die gleiche Menge an Fehlern finden wie eine Herangehensweise per brute-force, jedoch mit deutlich geringerem Zeitaufwand.
Leider wurde in der Arbeit [4] nicht drauf eingegangen, wie ScEpTIC arbeitet und wonach genau getestet werden kann. Maioli hat

in seiner Masterarbeit ScEpTIC deutlich detaillierter beschrieben und erklärt dort, wie genau das Tool funktioniert. Die folgenden Abschnitt stammt also nicht aus dem eigentlichen Papier, sondern aus der Masterarbeit von Maioli[3].

3.1 Anpassbarkeit der Architektur

ScEpTIC wurde so entwickelt, dass es unabhängig von der Architektur des Mikrocontroller arbeiten kann. Dies ermöglicht es den Code von unterschiedlichen Architekturen zu testen, ohne etwas am Code von ScEpTIC zu verändern oder den Code neu zu kompilieren[3].
Um dies zu erreichen verwendet ScEpTIC die folgenden Designentscheidungen:

- ScEpTIC verwendet LLVM IR Code, welcher ähnlich zu betrachten ist wie Assembler. Da wir auf dieser unteren Ebene der Programmierung arbeiten, wird nicht auf Elemente oder besondere Speicherbereiche einer Architektur referiert.
- Der Speicher wird soweit abstrahiert, dass nur zwischen volotile Memory und non volatile Memory unterschieden wird und besondere Bereiche wie der Stack und Heap nur von ScEpTIC selbst verwaltet wird.
- Die Checkpoint Logik musste überarbeitet werden, da die übliche Implementierung auf Referenzen zu Registern verwendet und somit Architektur abhängig ist.
- I/O Ports werden als Funktionen gesehen, wodurch der I/O Zugriff und Initialisierung ignoriert werden kann
- ScEpTIC stellt eine abstraktions Methode zur Verfügung, wodurch architekturabhängige Funktionalitäten als benutzerdefinierte Funktionen implementiert werden können.

Durch diese Designentscheidungen kann nun ScEpTIC ohne oder mit sehr kleinen Änderungen an andere oder neue Architekturen angepasst werden.

3.2 Anpassbarkeit des Checkpoint-Systems

ScEpTIC ist nicht nur Architektur unabhängig, sondern kann auch auf unterschiedliche Checkpoint Systeme konfiguriert werden. Dies ermöglicht es das Verfahren zu wählen, welches am besten zur Aufgabe der Mikrocontroller-Einheit passt. Wenn es für die Ausführung Beispielsweise wichtig ist, dass bei Variablen keine Inkonsistenzen entstehen, wenn der Strom genau in der Speicherung der Daten ausfällt, so würde sich DINO [1] empfehlen. DINO versioniert alle Variablen, wodurch genau dieser Form von Inkonsistenzen vorgebeugt werden kann.
Insgesamt gibt es fünf vordefinierte Checkpoint Systeme, welche sowohl statisch als auch dynamisch funktionieren. Jedoch ist es trotzdem möglich durch die Erstellung einer Python Datei jegliches Checkpoint-System in ScEpTIC einzuarbeiten [3].

3.3 Testumfang von ScEpTIC

Sobald ScEpTIC Konfiguriert wurde kann das zu testende Programm auf 4 unterschiedliche Arten analysiert werden.

- **Speicherinkonsistenzen**: ScEpTIC analysiert automatisch das Programm auf Speicherinkonsistenzen, wie den Data Access Bug, Activation Record Bug und Memory Map Bug. Sobald ScEpTIC die Untersuchung beendet hat gibt es eine Liste mit allen Inkonsistenzen aus.
- **Input Access Inkonsistenzen**: Damit ScEpTIC den Input Access analysieren kann benötigt es ein Modell des Input Access für jedes Eingabegerät. Danach wird vergleichen ob Eingaben dieses Modell erfüllen.
- **Profilierungsverfahren**: ScEpTIC verifiziert die Korrektheit der Eingaben im Programm selbst und die Interaktion mit externen Geräten.
- **Ausgabe Profilierungsverfahren**: ScEpTIC gibt aus, wann Ausgabeoperationen mehrfach ausgeführt werden, wodurch der Programmierer verifizieren kann, ob es zu Inkonsistenzen kommen kann.

ScEpTIC testet also teilweise automatisch und in anderen Fällen semi-automatisch, wo der Programmierer eingreifen muss oder die Ergebnisse selbst validieren muss.
Maioli et al. [4] geht in seiner Arbeit nur auf die Speicher-Inkonsistenzen ein, wodurch diese im weiteren Verlauf weiter betrachtet werden.

3.4 Testen der Speicher-Inkonsistenzen

ScEpTIC testet auf die vorher angeführten Bugs in Kapitel 2.1-2.3. Dabei fällt auf, dass alle Bugs eine ähnliche Ursache haben, nämlich einen Befehl vor einem Checkpoint, im Falle des Data Access Bugs einen Load Befehl, und einem Befehl, welcher bei der erneuten Ausführung zum Bug führt, als der Store Befehl. Somit muss nicht jede mögliche Checkpoint-Position getestet werden, wie es beim brute-force Ansatz der Fall wäre, sondern nur die Positionen an den kritischen Stellen.
Maioli et al. [4] beschreibt den Vorgang beim Data Access Bug wie folgt. Es muss nur getestet werden, wenn der Checkpoint genau vor dem Load Befehl liegt. Danach wird geschaut, wann der korrespondierende Store Befehl kommt und ob ein weiterer Checkpoint dazwischen liegt.
Nun kann zwar getestet werden ob ein Bug auftritt, jedoch nicht was dieser für eine Auswirkung hat. Dafür wird die kritische Stelle emuliert und die Ereignisse im NVM gespeichert. Somit kann erkannt werden, was für Ereignisse durch den aufgetretenen Bug entstehen.

4 SCEPTIC UND CHECKPOINT-SYSTEME

ScEpTIC ist ein reines analytisches Debug-Tool, welches Architektur und Checkpoint-System unabhängig funktioniert. Jedoch habe ich mir die Frage gestellt, ob ein Debug-Tool in diesen Ausmaßen überhaupt notwendig ist, wenn es Compiler und Checkpoint-Ansätze gibt, welche Inkonsistenzen vorbeugen. Hierzu habe ich mir genauer die Arbeitsweise von DINO [1] angeschaut. DINO stellt Ansätze zur Verfügung um Inkonsistenzen zu vermeiden. Dazu zählt zum einen die Verifizierung der Variablen, um Inkonsistenzen zu vermeiden, die durch Stromausfälle in der Speicherung von Variablen passieren, so wie die Analyse der Datenströme. DINO untersucht den Quellcode und legt so fest wann es zu Inkonsistenzen kommt, jedoch wird das Programm auf einer niedrigeren Abstraktionsstufe analysiert. Da jedoch nicht genau benannt wird, wie weit man abstrahiert kann es trotzdem zu Inkonsistenzen kommen. Deswegen ist es wichtig, so wie es bei ScEpTIC implementiert wurde, den Maschinencode selbst zu untersuchen [3].

DINO [1] analysiert auch nur die Speicherinkonsistenzen, übersieht jedoch Probleme bei der Interaktion mit der Umgebung, sowie Probleme im Stromverbrauch komplett.
Dazu kommt, das jeder Ansatz der Checkpoint-Systeme den Speicher anders konfiguriert, was auch zu Inkonsistenzen führen kann.
Somit wird deutlich, dass ein Analysetool definitiv gebraucht wird, um auch bei automatisch gesetzten Checkpoints die korrekte Ausführung zu bestätigen.

5 SCEPTIC'S STÄRKEN

ScEpTIC weißt einige Stärken auf, auf welche ich im folgenden eingehen werde.

5.1 Anpassbarkeit

ScEpTIC's wohl größte Stärke liegt in der Anpassbarkeit des Tools. Wie in Kapitel 3.1 angeführt kann ScEpTIC auf jede Architektur mit geringem Aufwand angepasst werden. Somit werden Entwickler nicht auf bestimmte Architekturen beschränkt und können diese ihrem Problem anpassen.
Durch die zur Verfügung gestellten Schnittstellen können weitere Funktionalitäten für die verwendetet Architektur schnell und sicher hinzugefügt werden, ohne im Quellcode von ScEpTIC zu arbeiten.

5.2 Erkennung von Bugs

ScEpTIC erkennt mehr Bugs als andere Tools, da nicht nur der Data Access Bug, sondern auch der Activation Record Bug, so wie der Memory Map Bug erkannt wird. Dies geschieht zwar nicht lückenlos, worauf in Kapitel 6 eingegangen wird, jedoch werden in den meisten Fällen alle Bugs gefunden, solange die Werte zum Testen gut gelegt werden. In so einen Fall findet ScEpTIC genauso viele Bugs wie das Testen durchs brute-forcen, welches jedoch deutlich länger dauern würde. Im Gegensatz dazu testet ScEpTIC in einer sehr annehmbaren Zeit, was die Verwendung des Tools in einer alltäglichen Situation möglich macht.

5.3 Erkennen von weiten Problemen

ScEpTIC priorisiert die Erkennung von Speicherinkonsistenzen, jedoch werden auch weitere Probleme betrachtet, wie die Überprüfung, ob die Übermittlung von Daten mit äußeren und inneren Prozessen ohne Probleme funktioniert.

6 SCEPTIC'S SCHWÄCHEN

ScEpTIC durchsucht Programme im statischen Code um ihren Effekt im laufenden Programm testen zu können. Dieses Prinzip funktioniert in den meisten Fällen sehr gut, da viele der zu prüfenden Bugs so gefunden werden können, jedoch gibt es Sonderfälle und Anwendungsfelder, wodurch Bugs übersehen werden können.
Dieses Problem tritt auf, wenn der Anwender eine execution Depth wählt, welche für das zu untersuchende Programm zu klein ist. Somit können die Muster, welcher erkannt werden sollen, übersehen werden. Als Beispiel nehme man ein Programm, welches einen Wert aus dem non volatile Memory lädt und nach n folgenden Befehlen diesen verändert und überspeichert. Bei einer execution Depth, welche kleiner als n ist, wird zwar das Laden des Wertes als Anfang eines möglichen Bugs erkannt, jedoch das eigentliche verändern und speichern des Wertes wird nicht mehr erkannt, da es "zu weit entfernt" ist.
Somit wird deutlich, dass ScEpTIC nicht alle Bugs findet und einen nicht definierbaren Teil übersehen könnte. Dies kann zu Problemen führen, da dieses Verhalten nicht einmal in dem Werk von Maioli erwähnt wurde und so einfach übersehen werden kann. Die Behauptung, dass ScEpTIC genau so viele Bugs wie ein brute-force-Ansatz findet, ist somit nicht richtig, da dieser genau diese Spezialfälle finden würde, im Gegensatz zu ScEpTIC.

7 FAZIT

ScEpTIC schafft es durch die systematische Analyse des zu testenen Programmes eine Vielzahl an Problemen zu erkennen, ohne dabei eine zeitliche Komplexität zu Besitzen, welche die Verwendung des Tools hinfällig machen würde. Auch die hohe Anpassbarkeit spielt eine Rolle, wodurch ScEpTIC Programme in der battery less IOT sicher machen könnte, wenn es richtig angewendet wird.
Da keine weiteren Arbeiten auf der Basis dieses Ansatzes bis zu diesem Zeitpunkt veröffentlicht wurden, kann leider nicht abgesehen werden, was für eine Auswirkung der Ansatz auf das Gebiet der battery less IOT schlussendlich haben wird.

REFERENCES

[1] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. *SIGPLAN Not.* 50, 6 (June 2015), 575–585. https://doi.org/10.1145/2813885.2737978
[2] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 96 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133920
[3] ANDREA MAIOLI. 2019. Understanding and Testing Intermittence Bugs in Transiently-powered Computers. *Proc. ACM Program. Lang.* (2019), 305. http://hdl.handle.net/10589/147444
[4] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2019. On Intermittence Bugs in the Battery-Less Internet of Things (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Phoenix, AZ, USA) *(LCTES 2019)*. Association for Computing Machinery, New York, NY, USA, 203–207. https://doi.org/10.1145/3316482.3326346
[5] Milijana Surbatovich, Brandon Lucia, and Limin Jia. 2020. Towards a Formal Foundation of Intermittent Computing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 163 (Nov. 2020), 31 pages. https://doi.org/10.1145/3428231
[6] Sumanth Umesh and Sparsh Mittal. 2021. A survey of techniques for intermittent computing. *Journal of Systems Architecture* 112 (2021), 101859. https://doi.org/10.1016/j.sysarc.2020.101859
[7] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 17–32.