Security Audits von Embedded Systems mit Mikrocontrollern

Markus Kammerstetter · Daniel Burian · Stefan Riegler

Trustworks KG
{m.kammerstetter | d.burian | s.riegler}@trustworks.at

Zusammenfassung

Eingebettete Systeme werden nicht nur immer breiter eingesetzt, sondern auch zunehmend vernetzt, wodurch ebenso die Gefahren von Hacker-Angriffen ansteigen. Besonders bei kritischen Systemen ist es für Anwender maßgeblich, herstellerunabhängige Security Audits durchzuführen um Schwachstellen vorzeitig erkennen und beheben zu können. Eine besondere Herausforderung stellen dabei Mikrocontroller mit internen Speichern dar, weil die enthaltene Firmware häufig nicht ausgelesen und folglich auch nicht auf Schwachstellen überprüft werden kann. In der vorliegenden Arbeit werden zuerst verbreitete Auslese-Schutzmaßnahmen von Standard-komponenten und -technologien vorgestellt. Anschließend werden Labor-Methoden beschrieben, die von unabhängigen Security Analysten eingesetzt werden können, um trotz Auslese-Schutz die Firmware zu extrahieren. Abschließend erfolgt ein Ausblick über Firmware-Sicherheits-Analyseverfahren.

1 Einleitung

Eingebettete Systeme werden heute allgegenwärtig eingesetzt. Aktuelle Studien gehen etwa von hunderten derartiger Systeme in unsrem direkten täglichen Umfeld aus [ArARS15]. Der Siegeszug von eingebetteten Systemen basiert dabei maßgeblich auf dem Einsatz von programmierbaren integrierten Schaltungen, wie Mikrocontroller. Erst durch Mikrocontroller werden kleine, energieeffiziente und kostengünstige Systeme ermöglicht. Ihre vielfältigen Einsatzgebiete reichen von tagtäglicher Unterhaltungs- und Haushaltselektronik, bis hin zu kritischen Anwendungsbereichen, wie medizinischen Geräten, Steuergeräten sowie Schließsystemen und Wegfahrsperren in Fahrzeugen, Komponenten in industriellen Produktionssystemen, Zutritts- und Schließsystemen oder den Feldkomponenten von kritischen Infrastrukturen. Studien gehen davon aus, dass der weltweite Markt für Mikrocontroller bis 2020 auf mehr als 50 Mrd. USD anwachsen wird [Gran15].

Besonders bei sicherheitskritischen, physischen Anwendungen können erfolgreiche Angriffe desaströse Auswirkungen mit sich bringen. In der Vergangenheit waren derartige Systeme oft kaum vernetzt, sodass deren Exponiertheit gegenüber Angreifern, aufgrund der fehlenden physischen Zugriffsmöglichkeit, beschränkt war. So waren etwa die Steuergeräte in Fahrzeugen auch schon früher über interne Bussysteme miteinander verbunden, jedoch gab es keine externen Schnittstellen, die enthaltene Systeme etwa zum Internet, zum Smartphone oder drahtlos zu anderen Geräten und Fahrzeugen vernetzten. Viele der heute im Einsatz befindlichen eingebetteten Systeme wurden somit mit einem Fokus auf Funktionalität und Betriebssicherheit entwickelt und deren Sicherheit gegenüber böswilligen Angriffen wurde etwa aufgrund der schwachen Vernetzung nur nebensächlich oder gar nicht berücksichtigt. Aktuell ist ein Paradig-

P. Schartner · A. Baumann (Hrsg.) · D•A•CH Security 2017 · syssec (2017) 38-49.

menwechsel im Gange, sodass Geräte zunehmend vernetzt und dadurch intelligenter werden. Man spricht von sogenannten Smart Devices, Industrie 4.0 und dem Internet der Dinge (IoT). Im Internet der Dinge werden zunehmend tagtägliche Geräte mit dem Internet verbunden. Viele Fahrzeuge haben bereits mehrfache Verbindungen zum Internet. Medizinische Geräte, wie Herzschrittmacher, lassen sich drahtlos parametrieren und auch Zutritts- und Schließsysteme bis hin zum Autoschlüssel lassen sich heute bequem drahtlos bedienen.

Auf der Kehrseite heißt dies, dass viele kritische sowie potenziell unsichere Systeme nun stark miteinander vernetzt und damit leichter von außen (etwa über Netzwerke oder drahtlose Schnittstellen) angreifbar werden. Es besteht somit aktuell eine Divergenz zwischen den hohen Sicherheitsanforderungen von eingebetteten Systemen einerseits und den dort enthaltenen oft unzureichenden Sicherheitsimplementierungen andrerseits [KZRJ13, ViTh12]. Wenngleich die Implementierung der Sicherheitsfunktionen bei den Herstellern der Systeme liegt, so sind es letztlich deren Kunden, die die Systeme einsetzen und im Falle von erfolgreichen Angriffen oftmals die Haftung übernehmen müssen. Für sicherheitskritische Anwendungen sind daher unabhängige Sicherheitsüberprüfungen in Form von Security Audits üblich. Sie können Schwachstellen aufdecken und ermöglichen einen tiefen sowie vom Hersteller unabhängigen Einblick in das Sicherheitsdesign und dessen Implementierung, über welchen das Vertrauen in eingesetzte Systeme gerechtfertigt werden kann.

Während heute für Software auf PC Systemen zahlreiche Analyseverfahren [AuWi11, LSCL12] von statischer und dynamischer Code-Analyse über Fuzz-Testing bis hin zu symbolischen oder konkolischen[1] Techniken [CARB12, ScAB10] genutzt werden können um Sicherheitslücken aufzudecken, stehen Sicherheitsanalysten bei Mikrocontrollern mit integrierten Speichern oft vor der Problemstellung, dass die enthaltene Firmware gar nicht erst ausgelesen werden kann und folglich auch keine Sicherheitsanalysen der Firmware möglich sind. Gleichzeitig stehen Angreifern üblicherweise mehr Zeit sowie fragwürdige Dienstleistungen (etwa von chinesischen Anbietern[2]) zur Verfügung um derartige Ausleseschutzmechanismen zu überwinden.

In den folgenden Abschnitten werden zuerst Speichertechnologien, Programmier- und Debug-Schnittstellen, sowie Ausleseschutzmechanismen von gängigen Mikrocontrollern mit integrierten Speichern behandelt. Darauf folgend werden Firmware-Extraktionstechniken beschrieben, die von Security Analysten in einer Labor Umgebung eingesetzt werden können, um Security Audits von Firmware zu ermöglichen. Abschließend erfolgt ein Ausblick über anwendbare Firmware-Analysetechniken, um Schwachstellen aufzudecken.

2 Related Work

Sowohl Khare et al. [KhSK11] wie auch Venkitaraman et al. [VeGu04] behandeln statische Code-Analyse von Embedded Systems. Feng et al. [FZXC+16] nutzen initial eine statische Code-Analyse von Firmware Images zur Extraktion von Control Flow Graphen (CFGs) in welchen anschließend automationsgestützt nach bekannten Schwachstellen gesucht werden kann. Zaddach et al. [ZBFB14], Costoni et al. [CoZF16] sowie Kammerstetter et al. [KaPK14] setzen hingegen auf dynamische Analyseverfahren um Schwachstellen mittels verbreiteter Analyseverfahren wie Fuzz-Testing bis hin zu Symbolic Execution aufzudecken. Sämtliche Methoden gehen jedoch stets davon aus, dass die Firmware bereits für die Analyse zur Verfügung steht.

[1] Kombination aus konkreten und symbolischen Techniken

[2] http://www.extract-ic.com, http://www.break-ic.com, http://www.ic-cracker.com, http://www.ic-crack.com

Skorobogatov [Skor05] behandelt nicht invasive, semi-invasive und invasive Methoden auf Mikrochips und die erzielten Ergebnisse stellen eine wesentliche Grundlage zur Firmware-Extraktion für diese Arbeit dar. Die Auslese-Schutzmechanismen von Mikrocontrollern werden allerdings nur in geringem Umfang angeschnitten. D. Strobel et al. [SORS+14] beschreiben mögliche Angriffe auf Embedded Systems mit Mikrocontrollern. Im Rahmen der Arbeit wird u.a. auf Firmware-Extraktionsmethoden für PIC Mikrocontroller eingegangen. Helfmeier et al. [HNTK+13] gehen speziell auf invasive Methoden mittels Focused Ion Beam (FIB) auf sichere Mikrocontroller wie Smartcards ein. Die vorliegende Arbeit konzentriert sich hingegen auf verbreitete Standardkomponenten und -technologien ohne spezielle Schutzmechanismen.

3 Ausleseschutz von Mikrocontrollern

Mikrocontroller mit internen, nicht volatilen Speichern wie Mask-ROMs oder Flash Speicher stellen unabhängige Security Analysten vor die Herausforderung, dass aufgrund von aktiven Ausleseschutzmaßnahmen (“Kopierschutz”) die Firmware nicht extrahiert und in weiterer Folge auch nicht auf Schwachstellen untersucht werden kann. Im Vergleich zu zeitlich limitieren Security Audits stehen Angreifern jedoch höhere zeitliche Ressourcen und fragwürdige Firmware-Extraktions-Dienstleistungen (vgl. Abschnitt 1) zur Verfügung. Eine einmalig entdeckte Schwachstelle und der dazugehörige Angriff können von ihnen im Internet veröffentlicht und so auch von weniger versierten Personen angewandt werden. Um den aktuellen Vorsprung von Angreifern zu verkleinern, benötigen unabhängige Security Analysten entsprechende Labor-Methoden um innerhalb eines vertretbaren Zeitaufwands Firmware trotz Ausleseschutz von Standardkomponenten und -technologien extrahieren und auf Schwachstellen überprüfen zu können. Sichere Mikrocontroller wie Smartcards enthalten hingegen zahlreiche Schutzmaßnahmen, die u.a. das Auslesen der enthaltenen Firmware wesentlich stärker absichern als dies Standardkomponenten und -technologien tun. Sie sind nicht Gegenstand der vorliegenden Arbeit und können von Angreifern nicht ohne erheblichen zeitlichen und finanziellen Aufwand mit Spezial Equipment wie aktuellen Focused-Ion-Beam-Geräten ausgelesen werden. Bevor jedoch auf Methoden zur Firmware-Extraktion von Standardkomponenten und -technologien eingegangen wird, werden im Folgenden die zu Grunde liegenden Speicher- und Programmierschnittstellen beschrieben.

3.1 Nicht volatile Speicher

Nicht volatile (d.h. nicht flüchtige) Speicher werden bei Mikrocontrollern eingesetzt, um Programmcode und notwendige Daten (wie z.B. Konfigurationsparameter oder Messwerte) zu speichern. Reine ROM Speicher wie Mask-ROMs erhalten ihren Speicherinhalt direkt bei der Chip-Fertigung durch den Halbleiter Hersteller. Enthaltener Code kann somit später nicht wieder geändert werden, weshalb diese Form von Speichertechnologie vorrangig für Anwendungen mit sehr hohen Stückzahlen ohne Updatewunsch (z.B. Autoschlüssel) oder für integrierte Bootloader (vgl. Abschnitt 3.2.2) verwendet werden. Wird eine Schwachstelle in einem Code-Teil im ROM identifiziert, so kann diese mangels Programmierbarkeit nicht behoben werden. Flash Speicher sind die gängigsten Speicher in Mikrocontrollern um Programmcode (wie Bootloader und Firmware) und statische Daten zu speichern [Skor05]. Flash-Speicher sind blockweise les- und programmierbar und können ggf. in einzelne Segmente unterteilt sein (etwa Bootloader- und Applikations-Segment). Auf EEPROMs kann byteweise zugegriffen werden. Sie werden üblicherweise zur Speicherung von Daten, nicht aber von Programmcode eingesetzt. Die jeweiligen Speicher sind üblicherweise auf unterschiedliche Adressbereiche gemappt (memory

mapping). Je nach Mikrocontrollertyp und -konfiguration können intern eingeschränkte Zugriffsrechte auf Speicher bestehen. Beispielsweise wäre es möglich, dass zwar Programmcode aus dem Mask-ROM Bootloader auf den Flash zugreifen darf, umgekehrt aber Programmcode im Flash-Speicher den Mask-ROM-Bootloader-Code nicht lesen darf.

3.2 Programmier- und Debug-Schnittstellen

Um nicht volatile Speicher wie EEPROM oder Flash Speicher programmieren zu können, existieren Programmierschnittstellen, wie die In-System-Programmierung (ISP) Schnittstelle, die auf serielle Kommunikationsinterfaces wie SPI, PDI oder JTAG (Abbildung 1 links) aufsetzen. Neben ISP unterstützen einige Mikrocontroller zudem herstellerabhängige Programmiermodi, wie etwa High-Voltage-Programmierung mit serieller oder paralleler Datenübertragung. Diese Programmiermodi sind üblicherweise in der Logik des Chips implementiert und können physisch über die Pins des jeweiligen Mikrocontrollers aktiviert werden. Neben physischen Programmierschnittstellen enthalten manche Mikrocontroller integrierte Bootloader. Die Bootloader sind nicht in der Chip Logik, sondern stattdessen in Software implementiert, die etwa in einem Mask-ROM oder in einem Segment des Flash-Speichers vorliegen kann. Sie implementieren typischerweise ein einfaches Kommunikationsprotokoll, welches über verbreitete Schnittstellen wie UART oder USB genutzt werden kann. Sowohl Bootloader wie auch physische Programmierschnittstellen stellen neben den Funktionen zum Löschen und Beschreiben von Speichern (z.B. Flash oder EEPROM) oft auch Funktionen zum Auslesen bzw. Verifizieren von Speicherblöcken zur Verfügung.

Neben Programmierschnittstellen existieren häufig Debug-Schnittstellen, die etwa bei der Entwicklung genutzt werden können, um Programmabläufe und Daten während des laufenden Betriebs des Mikrocontrollers zu überprüfen. Weitaus am häufigsten wird hier auf den JTAG-Standard [Asso13] gesetzt, dessen Herz die links in Abbildung 1 dargestellte Zustandsmaschine bildet. Um JTAG nutzen zu können, wird ein JTAG Dongle vgl. Abbildung 1 rechts benötigt. Der JTAG Dongle wird ähnlich zu einem typischen Programmiergerät verwendet: Eine Seite wird mit den JTAG Pins des Mikrocontrollers verbunden, während die andere über eine Software (wie z.B. einen architekturspezifischen gdb Debugserver) angesteuert wird.

Sämtliche enthaltene Programmier- und Debug-Schnittstellen sind für Security Analysten insofern maßgeblich, weil sie zumeist Funktionen enthalten mittels welchen die internen Speicher entweder direkt oder über Umwege ausgelesen werden können. Gleichermaßen setzen hier die von den Chip Herstellern eingesetzten Kopierschutzmaßnahmen an.

3.2.1 Security Fuse

Mikrocontroller lassen sich häufig über sogenannte Fuses konfigurieren. Hier kann beispielsweise konfiguriert werden, ob der interne oder der externe Oszillator verwendet werden oder wie hoch die interne Taktrate sein soll. Es existieren unterschiedliche technische Umsetzungen von Fuses, die über das Programmiergerät gesetzt werden können. Echte Fuses sind tatsächlich Einmal-Verbindungen, die durch das Programmieren physisch durchgebrannt und von daher nicht wieder programmatisch zurückgesetzt werden können. Der Großteil der Fuses kann jedoch einfach als nichtflüchtig programmierbares Register betrachtet werden.

Für Security Analysten sind vor allem die Security Fuses relevant. Sie definieren, ob beispielsweise Speicher über Programmierschnittstellen ausgelesen werden dürfen oder ob die JTAG Schnittstelle für das Debuggen genutzt werden kann. Wenngleich Security Fuses üblicherweise

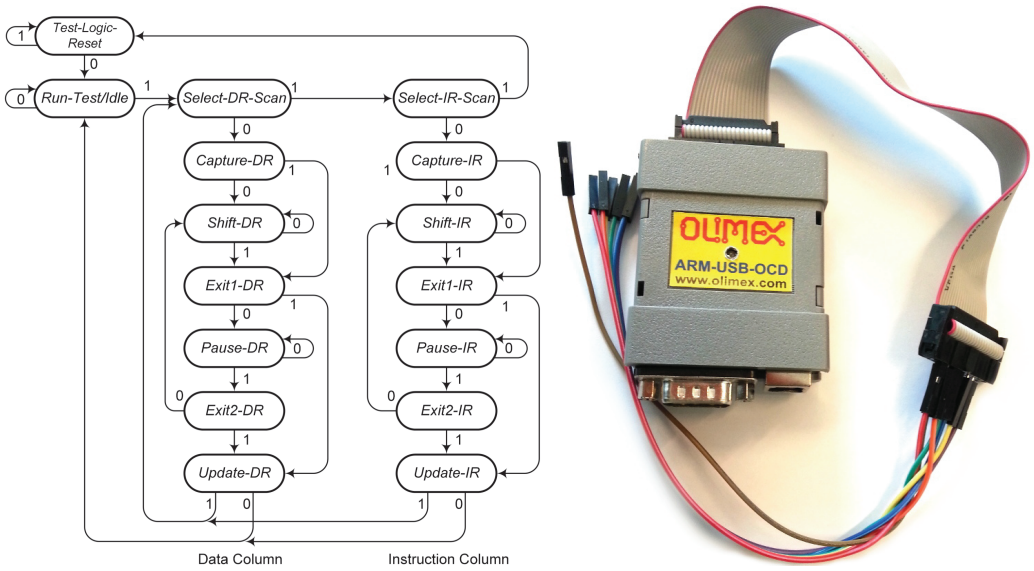

Abb. 1: Links: JTAG Zustandsdiagramm [Asso13], rechts: JTAG Dongle

zurückgesetzt werden können, so verursacht das Zurücksetzen intern beispielsweise die Löschung aller zuvor geschützten Speicherinhalte. Bei den später behandelten Extraktionsmethoden gilt es somit Maßnahmen zu finden um trotz gesetzter Security Fuses an die Speicherinhalte zu gelangen.

3.2.2 Bootloader Schutzmaßnahmen

Bootloader sind nicht in der Chip-Logik, sondern stattdessen in Software implementiert, die sich beispielsweise in Mask-ROM oder Flash-Speichersegmenten befindet. Beim Hochfahren des Mikrocontrollers wird zuerst der Bootloader Code ausgeführt. Trifft keine Bedingung zu in den Bootloader-Modus zu wechseln, so springt der Bootloader schließlich in das programmierte Anwendungsprogramm (Firmware). Wird jedoch der Bootloader-Modus aktiviert, so kann der Nutzer etwa über UART oder USB mit dem Bootloader kommunizieren. Bootloader können zahlreiche Funktionen unterstützen, wobei neben dem Programmieren zumeist auch Funktionen zum Auslesen enthalten sind. Bei Bootloadern sind die Schutzmechanismen üblicherweise in Form von Bootloader-Passwörtern oder Flags umgesetzt. Sobald ein Passwort gesetzt ist, stehen ohne vorige Authentifizierung nur mehr eingeschränkte Funktionen wie beispielsweise das Zurücksetzen des Passworts (und damit auch das Löschen der enthaltenen Speicherinhalte) zur Verfügung. Mittel Flags kann beispielsweise, ähnlich zu einer Security Fuse, die Auslesefunktion deaktiviert werden. Wird das Flag zurückgesetzt, so werden damit ebenso die enthaltenen Speicher gelöscht. Später beschriebene Extraktionsmethoden für Security Analysten zielen daher darauf ab, entweder das Bootloader Passwort zu eruieren, die Prüfung des Passworts bzw. des Flags zu umgehen oder beim Zurücksetzen den Löschvorgang der Speicher zu verhindern.

3.3 Verschlüsselte Firmware

Verschlüsselte Firmware kann bei Mikrocontrollern an mehreren Stellen zum Einsatz kommen. Befindet sich auf dem Mikrocontroller ein Bootloader, so könnte dieser etwa eine Firmware-Update-Funktion enthalten, sodass nur vom Hersteller signierte sowie verschlüsselte Firmware Updates eingespielt werden können. Ebenso könnte ein leicht auslesbarer externer Speicherbau-

stein zum Einsatz kommen, in welchem die Firmware verschlüsselt (und ggf. signiert) abgelegt ist. In beiden Szenarien kann die verschlüsselte Firmware von Security Analysten entweder aus einem Firmware Update oder aus dem externen Speicherbaustein ausgelesen werden, jedoch ist sie ohne den für die Entschlüsselung notwendigen Schlüssel nicht analysierbar. Letztendlich muss jedoch der Mikrocontroller den Schlüssel kennen und diesen somit in einem nicht volatilen internen Speicher hinterlegt haben. Bei der Extraktion des Schlüssels können somit bei Standardkomponenten und -technologien ähnliche Verfahren wie auch zur Firmware-Extraktion zum Einsatz kommen. Sichere Mikrocontrollern wie Smartcards beinhalten hingegen wesentlich höhere Schutzmaßnahmen und sind dediziert nicht im Fokus dieser Arbeit.

4 Firmware-Extraktionsmethoden

Im Folgenden werden unterschiedliche Methoden vorgestellt, die sich für die Firmware-Extraktion aus Mikrocontrollern einsetzen lassen. Die Methoden lassen sich allgemein in nicht-invasive, semi-invasive und invasive Methoden [Skor05] einordnen. Nicht-invasive Methoden sind Methoden, bei welchen das Gehäuse des Mikrocontroller-Chips nicht geöffnet werden muss. Hierzu zählen vor allem logische sowie Seitenkanal- und Fault-Injection-Angriffe, die ohne hohen Ausstattungs- und Kostenaufwand umgesetzt werden können. Bei semi-invasive Angriffen wird das Bauteil geöffnet, sodass freier Blick auf den enthaltenen Siliziumchip besteht. Bei invasiven Angriffen wird der im Gehäuse enthaltene Chip geöffnet und/oder modifiziert. Wenngleich diese Methoden bei Standard-Mikrocontrollern und -technologien leicht innerhalb eines beschränkten Zeitrahmens umsetzbar sind, so erfordern sie eine erhebliche technische Ausstattung.

4.1 Seitenkanalangriffe

Seitenkanalangriffe eignen sich vor allem zum Auffinden von Bootloader Passwörtern. Ein Bootloader-Passwort-Check vgl. Listing 1 ist beispielsweise anfällig für eine einfache Zeitanalyse, weil die Laufzeit der Passwort Überprüfung nicht konstant ist. Der Passwortcheck retourniert, sobald das erste Zeichen nicht mit dem gespeicherten Bootloader-Passwort übereinstimmt. Es können folglich alle Zeichen an einer Stelle durchiteriert werden, bis ein Kandidat gefunden wird, für welchen der Passwortcheck geringfügig länger benötigt. In diesem Fall ist ein richtiges Passwort-Zeichnen identifiziert und es kann mit dem nächsten Zeichen fortgefahren werden. Der Vorgang wird wiederholt, bis das Passwort erfolgreich rekonstruiert wurde.

```
bool check_bootloader_password(char *passwd)
{
        for (int i=0; i<PASSWD_LEN; i++) {
                if (passwd[i] != stored_BL_passwd[i])
                        return false;
        }
        return true;
}
```

Listing 1: Beispiel für einen anfälligen Passwortcheck mit nicht konstanter Laufzeit

Leistungsfähigere Seitenkanalangriffe nutzen hingegen den Stromverbrauch oder die elektromagnetische Abstrahlung des Mikrocontrollers. Ohne tiefer gehender auf die umfassende Thematik von Seitenkanalangriffen [SORS+14, Skor05] einzugehen, ist die wesentliche Beobachtung, dass die beim Bootloader-Passwortcheck ausgeführten CPU-Instruktionen datenabhängig

(und daher auch vom gespeicherten Passwort abhängig) sind. Beispielsweise kann so nicht nur eine in einer Schleife ausgeführte `compare` Instruktion anhand ihres Stromverbrauchs bzw. ihrer Abstrahlung identifiziert werden, sondern es ergeben sich ebenso datenabhängige Variationen die einen direkten Rückschluss auf die einzelnen Bytes des Passworts zulassen. Mittels mehrerer Messungen, Mittelung und statistischer Methoden wie der Berechnung der Korrelation (Correlation Power Analysis) lassen sich die Erfolgschancen weiter erhöhen.

4.2 Fault-Injection-Angriffe

Die Grundidee von Fault-Injection-Angriffen ist es, während des Betriebs des Mikrocontrollers absichtlich und gezielt einen kurzzeitigen Logik-Fehler zu verursachen, indem etwa ein externes Clock-Signal oder die Versorgungsspannung kurzzeitig variiert wird. Die CPU des Mikrocontrollers sowie die Logik zum Auslesen von Security Fuses oder Speichern sind üblicherweise mittels synchroner Register Transfer Logik (RTL) vgl. Abbildung 2 aufgebaut. Daten sind in einem Register zwischengespeichert und warten auf das Taktsignal. Sobald dieses eintrifft, wandern die Daten durch einen kombinatorischen Logik Block bis sie schließlich im Zielregister angekommen und der Vorgang von Neuem stattfinden kann. Dabei sind zwei Beobachtungen ausschlaggebend: (1) Der Durchlauf der Daten, durch die in der kombinatorischen Logik enthaltenen Gatter, benötigt Zeit. Je zahlreicher und komplexer die Gatter, desto mehr Zeit wird benötigt bis am Ausgang das korrekte Ergebnis anliegt. (2) Sämtliche Logikblöcke in einem Chip arbeiten parallel. Die beiden Umstände lassen sich nun beispielsweise von einem Security Analysten ausnutzen, indem etwa zum Zeitpunkt einer Ausleseschutzprüfung ein kurzzeitig zu schnelles Taktsignal verwendet wird (“Clock Glitching”). Der Instruktionspointer der CPU wird dadurch aufgrund seiner einfachen (und damit schnellen) Logik bereits erhöht, bevor der Befehl für den bedingten Sprung beim Passwortcheck überhaupt korrekt dekodiert werden konnte. Effektiv lässt sich so beispielsweise ein Passwortcheck in einem Bootloader überspringen.

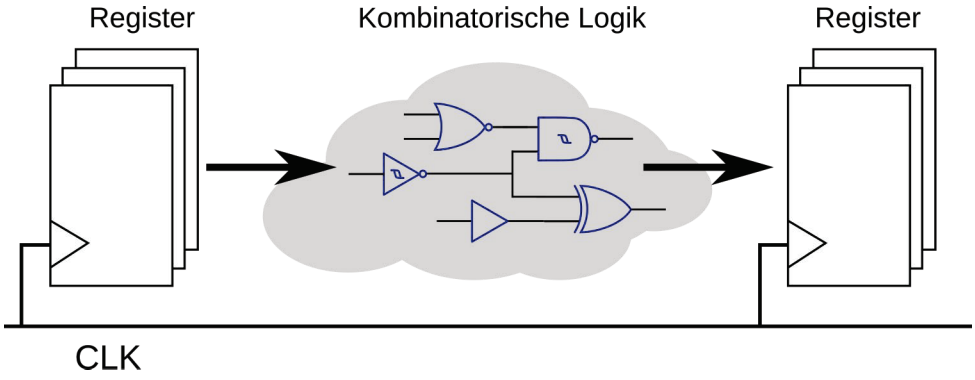

Abb. 2: Exemplarische Darstellung von Register Transfer Logik (RTL)

Ähnliche Effekte lassen sich erzielen indem kurzzeitig die Versorgungsspannung variiert wird (“Voltage Glitching”). Gängige Speichertechnologien benötigen intern Referenzspannungen um zu entscheiden, ob ein Bit gesetzt oder nicht gesetzt ist. Wird kurzzeitig die Versorgungsspannung verändert, so ändert sich auch die Referenzspannung und Speicherinhalte (wie ein Ausleseschutz-Bit) werden ggf. nicht korrekt ausgelesen. Obwohl der Ausleseschutz gesetzt ist, kann somit z.B. über Programmier- und Debug-Schnittstellen die enthaltene Firmware ausgelesen werden. Selbst wenn keine Programmier- oder Debug-Schnittstellen existieren, lassen sich ggf. in der Firmware enthaltene Kommunikationsroutinen (etwa für USB oder UART Schnitt-

stellen) dahingehend beeinflussen, dass sie zu viele Daten ausgeben. In Anbetracht des typischerweise kleinen Adressraumes kann so ggf. mittels eines Überlaufs die gesamte Firmware extrahiert werden. Ähnlich zur Thematik der Seitenkanalangriffe, ist auch der Themenbereich der Fault-Injection-Angriffe sehr umfassend [Skor05].

4.3 Semi-invasive Methoden

Bei semi-invasiven Methoden wird das Gehäuse des Mikrocontrollers typischerweise mittels nasschemischer Methoden geöffnet, sodass der enthaltene Siliziumchip entweder von der Vorder- bzw. von der Rückseite einsehbar und noch voll funktionsfähig ist. Security Fuses befinden sich häufig örtlich außerhalb der Bereiche für Flash und EEPROM Speicher. Bei dem in Abbildung 3 links ersichtlichen Mikrocontroller ist der Fuse Bereich links unten ersichtlich, während sich der Flash und der EEPROM Speicher eher in der Mitte befinden. Sind enthaltene Metall-Abschirmungen nicht vorhanden, zu dünn oder zu klein gewählt, so lassen sich die Fuses je nach Fuse-Technologie ggf. mittels UV Laser löschen. Für derartige Arbeiten können etwa motorisierte Mikroskope vgl. Abbildung 3 rechts eingesetzt werden. Nachdem einzelne Transistoren(-gruppen) durchschalten, wenn Licht auf ihr Gatter dringt (Fototransistor Effekt), kann fokussiertes Laserlicht auch gezielt für Fault-Injection-Angriffe eingesetzt werden, um beispielsweise die Auslese-Logik von Security Fuses zu beeinflussen. Nachdem jedoch auf der Oberseite des Chips üblicherweise mehrere Metallschichten das Laserlicht blockieren und sich die eigentlichen Transistoren ausschließlich in der untersten Schicht befinden, besteht die Möglichkeit entsprechende Angriffe von der Rückseite aus durchzuführen. Dabei kommen vor allem Infrarot-Laser zum Einsatz, weil das Träger-Silizium in diesem Wellenlängenbereich transparent ist. Eine umfassende Beschreibung semi-invasiver Methoden ist in [Skor05] zu finden.

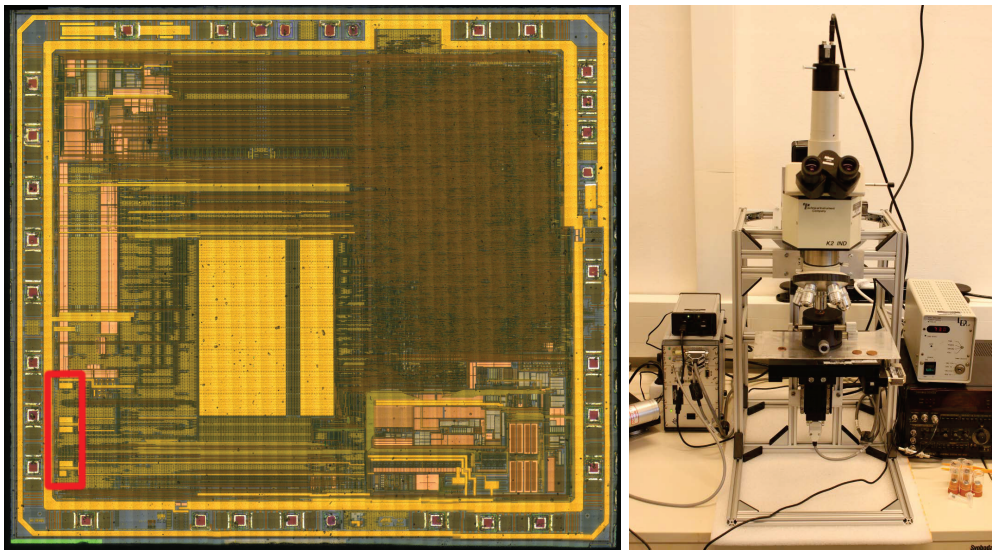

Abb. 3: Links: Mikrocontroller Fuse Bereich, rechts: Automatisiertes Konfokalmikroskop

4.4 Invasive Methoden

Invasive Methoden ermöglichen den direkten physischen Eingriff in den Mikrocontrollerchip. Sie stellen eine der mächtigsten Methoden dar, erfordern jedoch auch ein gut ausgestatte-

tes Hardware-Security-Labor mit Großgerätschaften wie Rasterelektronenmikroskop, Focused Ion Beam Workstation und diversen Gerätschaften zur Probenpräparation. Steht die notwendige Ausstattung zur Verfügung, so können übliche Schutzmechanismen auf Standardkomponenten und -technologien innerhalb eines absehbaren Zeitraums gut umgangen werden um beispielsweise die Firmware zu extrahieren. Sofern weniger invasive Methoden fehlschlagen, ermöglichen insbesondere elektronen-optische Verfahren die Extraktion von Firmware-Inhalten aus Mask-ROM-Speichern. In Abbildung 4 links ist etwa ein Ausschnitt des ROM-Inhalts eines verbreiteten Autoschlüssel-Mikrocontrollers ersichtlich. Die einzelnen Bits sind gut zu sehen, sodass der gesamte Firmware-Inhalt mittels eines einfachen Softwaretools rekonstruiert und folglich analysiert werden kann. Im konkreten Fall ist beispielsweise der gesamte Algorithmus in der Firmware enthalten, der in der Übertragung zwischen Autoschlüssel und Auto eingesetzt wird. Neben elektronen-optischen Firmware-Extraktionsverfahren können mittels einer Focused Ion Beam Workstation (Abbildung 4 rechts) einzelne Teile des Mikrochips modifiziert werden. Damit ist es etwa mit relativ kleinem Aufwand möglich, eine Security Fuse zu überbrücken, sodass anschließend etwa über eine Programmier- oder Debug-Schnittstelle die Firmware extrahiert werden kann. Durch die Editierbarkeit sind jedoch auch weitaus mächtigere Methoden, wie das direkte Auslese von Speichern möglich. Hierbei werden etwa interne Speicherbusse über kleine Probe Pads nach außen kontaktiert. Die Probe Pads können dann mit Nadeln kontaktiert und so zu Standardmess- und -prüfgeräten wie Oszilloskop und Logikanalysator verbunden werden.

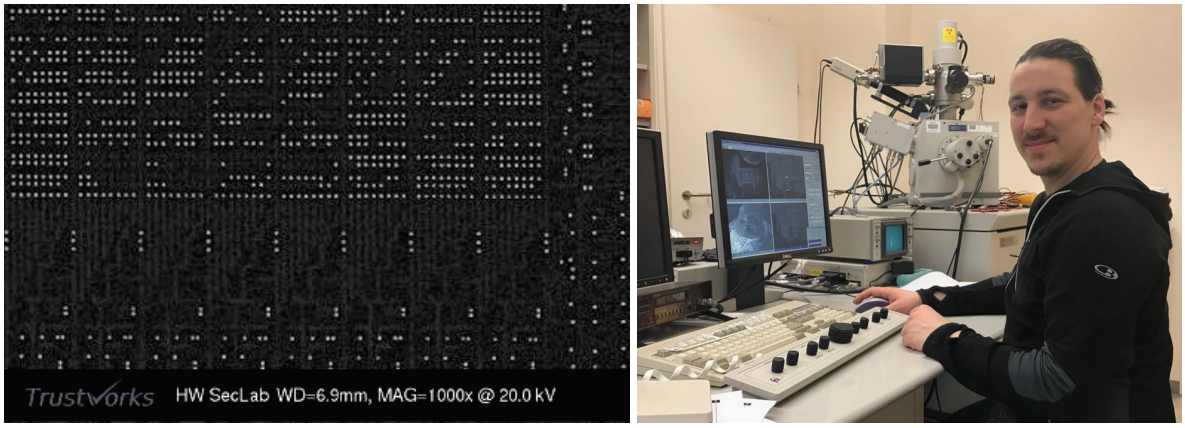

Abb. 4: Links: MaskROM Speicherinhalt (Autoschlüssel), rechts: Fei FIB 200 Workstation

5 Firmware-Analyseverfahren

Steht die Firmware bereits zur Verfügung oder wird sie mit Hilfe der vorher genannten Methoden (Abschnitt 4) aus dem Controller extrahiert, so kann diese weiter untersucht werden. Hierbei wird üblicherweise besonderes Augenmerk auf Schwachstellen in der Implementation (z.B. Overflow-Schwachstellen) sowie auf die sichere Anwendung kryptographischer Verfahren und Schlüssel gelegt. Sollte die Firmware einen größeren Funktionsumfang haben und ggf. ein Betriebssystem beinhalten, so kann es erforderlich sein, die Firmware erst zu entpacken um deren einzelne Bestandteile dann gesondert zu untersuchen. Allgemein werden zwei unterschiedliche Verfahren der Code Analyse unterschieden: statische und dynamische Codeanalyse, auf welche in den folgenden Unterabschnitten näher eingegangen wird. Beide Analysemethoden können sich auch gegenseitig unterstützen.

5.1 Statische Codeanalyse

Nach der Firmware-Extraktion liegt der Code in Binärform vor. Im Gegensatz zur dynamischen Analyse wir der Code nicht ausgeführt bzw. nicht während der Laufzeit analysiert. Vorteil der statischen Analyse ist, dass keine besondere Umgebung notwendig ist, um den Code auszuführen oder zu instrumentieren. Es können leistungsfähige Disassembler und Decompiler wie Ida Pro[3] eingesetzt werden, sofern sie die jeweilige Architektur unterstützen. In einem ersten Schritt werden üblicherweise sicherheitsrelevante Datenquellen (wie z.B. Lesefunktionen von Kommunikationsschnittstellen) identifiziert. Anschließend kann die umgesetzte Programmlogik und der Programmcode mit besonderem Augenmerk auf logische Fehler oder Programmierfehler untersucht werden. Ebenso können automationsgestützt sicherheitsrelevante Daten wie kryptografische Zertifikate, Passwörter oder Strings ausgelesen und analysiert werden. Der Nachteil von statischer Analyse ist, dass Laufzeitinformationen (wie etwa Werte von Variablen oder Zeigern) nicht zur Verfügung stehen und komplexe Abläufe ohne konkrete Verarbeitung bekannter Testdaten nur schwer nachvollzogen werden können.

5.2 Dynamische Codeanalyse

Die dynamische Analyse unterscheidet sich zur statischen Analyse dahingehend, dass der Firmwarecode entweder direkt auf der Ziel-Hardware oder in einem Emulator ausgeführt und somit während der Laufzeit analysiert werden kann. Dies ist möglich, wenn eine Debug-Schnittstelle wie JTAG (siehe Abschnitt 3.2) oder ein Emulator für die gegebene Plattform zur Verfügung steht. Der große Vorteil der dynamischen Analyse ist vor allem der, dass Laufzeitinformationen zur Verfügung stehen und der Programmablauf während der Ausführung analysiert und ggf. sogar modifiziert werden kann. Dadurch können Daten, Funktionen oder kryptographische Schlüssel, die eventuell erst während der Laufzeit entschlüsselt oder erzeugt werden, effizient analysiert und getestet werden. Zudem ist ein Embedded System oft mit vielen peripheren Komponenten verbunden, welche in einer statischen Analyse kaum oder gar nicht berücksichtigt werden können. Dies ist auch in dem Fall eines Emulators relevant, da externe Systeme üblicherweise nicht in einem Emulator umgesetzt sind. Hierbei spielen sich die Stärken des Debuggens mittels JTAG aus, da in dem Fall direkt auf der echten Hardware gearbeitet werden kann. Eine gut aufgebaute Debugumgebung ermöglicht zudem effiziente automationsgestützte Testverfahren wie Fuzz-Testing, bei welchen automatisch Testdaten generiert und von der Firmware verarbeitet werden. Tritt bei der Verarbeitung ein Fehler bei der Programmausführung auf, so kann dieser durch das Fuzz-Testing Tool mit Hilfe des Debuggers detektiert werden. Der Testfall, der zu einem Fehler führte, kann folglich mittels manueller Analyse genauer untersucht werden.

6 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurden übliche Mikrocontroller-Schutzmechanismen vorgestellt, die in gängigen Standardkomponenten und -technologien als Kopierschutz eingesetzt werden. Dieselben Mechanismen erschweren jedoch heute notwendige Security Audits von Embedded Systems, da die enthaltene Firmware nicht innerhalb eines praktikablen Zeitraums extrahiert und unabhängig auf Schwachstellen überprüft werden kann. Angreifern stehen hingegen üblicherweise nicht nur mehr Zeit um Schwachstellen z.B. auch ohne Firmware zu finden, son-

[3] https://www.hex-rays.com

dern auch fragwürdige Extraktions-Services im Ausland zur Verfügung. Mittels den vorgestellten Extraktions-Methoden wird es Security Analysten erleichtert, die Firmware in-house mit Laborgerätschaften selbst zu extrahieren. Erst dadurch werden unabhängige Security Audits von Embedded Systemen mit Mikrocontrollern und internen Speichern innerhalb eines praktikablen Zeitraums durchführbar.

Literatur

[ArARS15] R. Araujo, E. Anjos, D. Rousy Silva: Trends in the Use of Design Thinking for Embedded Systems. *In: Computational Science and Its Applications (ICCSA), 2015 15th International Conference on* (2015), 82–86.

[Asso13] T. I. S. Association: IEEE Standard for Test Access Port and Boundary-Scan Architecture. In: *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)* (2013), 1–444.

[AuWi11] A. Austin, L. Williams: One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques. *In: Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on* (2011), 97–106.

[CARB12] S. K. Cha, T. Avgerinos, A. Rebert, D. Brumley: Unleashing Mayhem on Binary Code. *In: Security and Privacy (SP), 2012 IEEE Symposium on* (2012), 380–394.

[CoZF16] A. Costin, A. Zarras, A. Francillon: Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. *In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, ACM, New York, NY, USA (2016), 437–448, http://doi.acm.org/10.1145/2897845.2897900.

[FZXC+16] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, H. Yin: Scalable Graph-based Bug Search for Firmware Images. *In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, ACM, New York, NY, USA (2016), 480–491, http://doi.acm.org/10.1145/2976749.2978370.

[Gran15] I. Grand View Research: Microcontroller Market Analysis By Product (8-bit, 16-bit, 32-bit), By Application (Automotive, Consumer Electronics, Industrial, Medical Devices, Military & Defense) And Segment Forecasts Till 2024 (2015), http://www.grandviewresearch.com/industry-analysis/microcontroller-market.

[HNTK+13] C. Helfmeier, D. Nedospasov, C. Tarnovsky, J. S. Krissler, C. Boit, J.-P. Seifert: Breaking and Entering Through the Silicon. *In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, ACM, New York, NY, USA (2013), 733–744, http://doi.acm.org/10.1145/2508859.2516717.

[KaPK14] M. Kammerstetter, C. Platzer, W. Kastner: Prospect: Peripheral Proxying Supported Embedded Code Testing. *In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, ACM, New York, NY, USA (2014), 329–340, http://doi.acm.org/10.1145/2590296.2590301.

[KhSK11] S. Khare, S. Saraswat, S. Kumar: Static Program Analysis of Large Embedded Code Base: An Experience. *In: Proceedings of the 4th India Software En-*

gineering Conference, ISEC '11, ACM, New York, NY, USA (2011), 99–102, http://doi.acm.org/10.1145/1953355.1953368.

[KZRJ13] M. Kermani, M. Zhang, A. Raghunathan, N. Jha: Emerging Frontiers in Embedded Security. *In: VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on* (2013), 203–208.

[LSCL12] B. Liu, L. Shi, Z. Cai, M. Li: Software Vulnerability Discovery Techniques: A Survey. *In: Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on* (2012), 152–156.

[ScAB10] E. Schwartz, T. Avgerinos, D. Brumley: All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). *In: Security and Privacy (SP), 2010 IEEE Symposium on* (2010), 317–331.

[Skor05] S. P. Skorobogatov: Semi-invasive attacks – A new approach to hardware security analysis. Tech. Rep. UCAM-CL-TR-630, University of Cambridge, Computer Laboratory (2005), http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.pdf.

[SORS+14] D. Strobel, D. Oswald, B. Richter, F. Schellenberg, C. Paar: Microcontrollers as (In)Security Devices for Pervasive Computing Applications. In: *Proceedings of the IEEE*, 102, 8 (2014), 1157–1173.

[VeGu04] R. Venkitaraman, G. Gupta: Static Program Analysis of Embedded Executable Assembly Code. *In: Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '04, ACM, New York, NY, USA (2004), 157–166, http://doi.acm.org/10.1145/1023833.1023857.

[ViTh12] J. Viega, H. Thompson: The State of Embedded-Device Security (Spoiler Alert: It's Bad). In: *Security Privacy, IEEE*, 10, 5 (2012), 68–70.

[ZBFB14] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti: AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. *In: NDSS*, The Internet Society (2014), http://dblp.uni-trier.de/db/conf/ndss/ndss2014.html#ZaddachBFB14.