Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Gabriel Capella

Título da monografia
se for longo ocupa esta linha também

São Paulo
Dezembro de 2018

Título da monografia
se for longo ocupa esta linha também

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Alfredo Goldman vel Lejbman

São Paulo
Dezembro de 2018

Resumo

Elemento obrigatório, constituído de uma sequência de frases concisas e objetivas, em forma de texto. Deve apresentar os objetivos, métodos empregados, resultados e conclusões. O resumo deve ser redigido em parágrafo único, conter no máximo 500 palavras e ser seguido dos termos representativos do conteúdo do trabalho (palavras-chave).

Palavras-chave: palavra-chave1, palavra-chave2, palavra-chave3.

Abstract

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa.

Keywords: keyword1, keyword2, keyword3.

Contents

Chapter 1

Introdução

falar que atuamente varios sao vulneráveis
pegamos uma solucao, melhoramos, testamos e vimos que eh possivel

Chapter 2

Key Concepts

Nowadays the term IoT (Internet of Things) is turning a buzzword. People, government, and industry have learned that connect simple devices to the internet, can produce a lot of useful data. This data can be used to improve production, make houses more smart e save money and time. Some studies show that by 2020 will be more than 20 billion IoT devices connected to the internet [].

This huge number of devices create new safety and security challenges. But the wide number of known attacks in the lasts years show the responsible for this devices is not taking this challenge seriously. Some of this attacks can be in seen in the article X. Some of this attacks occur inside critical infrastructures, like nuclear plants, health, and transportation system.

Software, network, and hardware are layers of this devices that can be attack and damage. This work is dedicated to studying technics to verify if one device is attacked. Attacks will ever occur, but if it occurs must have a way to identify than.

2.1 Embedded Devices

One embedded device is a system build to make a specific task. The main difference from normal circuits is the fact that this hardware is programmable, making possible the changes of it functionality only change the program inside it. Normal this devices are low cost, against failures and build work in real time situations. Because they aren't built for general propose, in the most cases they don't have a full operating system running inside it.

Nowadays these devices are changing, they are being connected to the to the internet and making part of the Internet of Things. This makes them vulnerable to attacks and invasions. Make them safe without increasing their cost and availability are challenges.

2.2 Root of Trust

One of the principal themes related to security is the Root of Trust. Suppose Alice and to talk to Bob on an encrypted channel. They can use a symmetric or an asymmetric encryption algorithm to do that. If they choose a symmetric algorithm before they start talking they need to share a password. If they choose an asymmetric algorithm, they need to share their public key in a safe way. If they share their public keys in an insecure channel, will not be possible to prove the identity of the other. There is a need for a reliable channel to start the conversation, this idea is similar of the Root of Trust idea.

Another example of Root of Trust can be seen in the TLS (Transport Layer Security) use. When an encrypted connection is made, they need to use previous saved public keys (certificates). These certificates are provided during the operating system installation. There is a belief that the system is not corrupted during the installation and because of that, the certificates are not modified. The root of trust, in this case, are made by the installation moment.

Besides cryptography, the idea of the root of trust can be used in software verification. Suppose you want to know if a specific software is running on a device, you can verify the software before the run and, if it is the original one, you run with. On practical example is a computer manufacturer who wants their machines to run a specific version of an operational system. They can program the bootloader to verify the executable before running it. The root of trust, in this case, is made when the bootloader is written.

In field software verification are two roots of trust types[7]: dynamic and static. In the static one, the verification is only made before the software execution. In the dynamic, it is made during the software execution.

2.3 Remote Attestation

minimalist aproch
Hardware x Software (TCP)

2.4 Works in Remote Attestation for Embedded Devices

One of the main remote attestation projects using the software approach is Piooner.

Currently the main works done in this area are: SMART [6], TrustLite [8] and SANCUS [9].

SPM (software protection modules, Pionner), TrustLite, SMART, SANCUS.

2.5 FPGA

FPGA is an abbreviation for "field-programmable gate array". This is a special type of integrated circuit created in the 80s, the main difference of others integrates circuits is the possibility to reprogram the circuit using an HDL (hardware description language).

The major benefits are that you can test new hardware and circuits without the need to produce a new integrated circuit. One of the tasks in this work is to change the memory backbone of on microcontroller and test the results. Using the FPGA the assignment to test the modification became easy because changing it is like reprogram.

There are two principal HDL languages used in the market: Verilog and VHDL. This language is totally differences from normal programming languages. The principal difference from normal programming languages is the fact that everything is happening in parallel. In this languages, variables are replaced by registers and connected with wires. There are some attempts to convert sequential programming languages, like C, in hardware description languages, like the LegUp High-Level Synthesis project [3]. But transforming sequential code to hardware normally require big circuit.

An approach used in used in most FPGA projects is to program a small processor inside it. This processor can be programmed using a sequential programming language. The good thing about this designer is the possibility to connect an hardware in the processor to make

specifics tasks. Some companies that produce FPGA's are build processor optimized for their products. One of the most famous projects is MicroBlaze, a 32-bit RISC microprocessor, designed by Xilinx. A Linux kernel implementation has been made for this processor, making possible to run a Linux inside an FPGA.

2.6 Layers Contemplated

This work will build a device and program it, involves a different abstraction layer. The first layer, already explained, is the FPGA. It will run a hardware description code and will build this hardware inside it. This hardware will run an assembly code generated by a compiler from the C language.

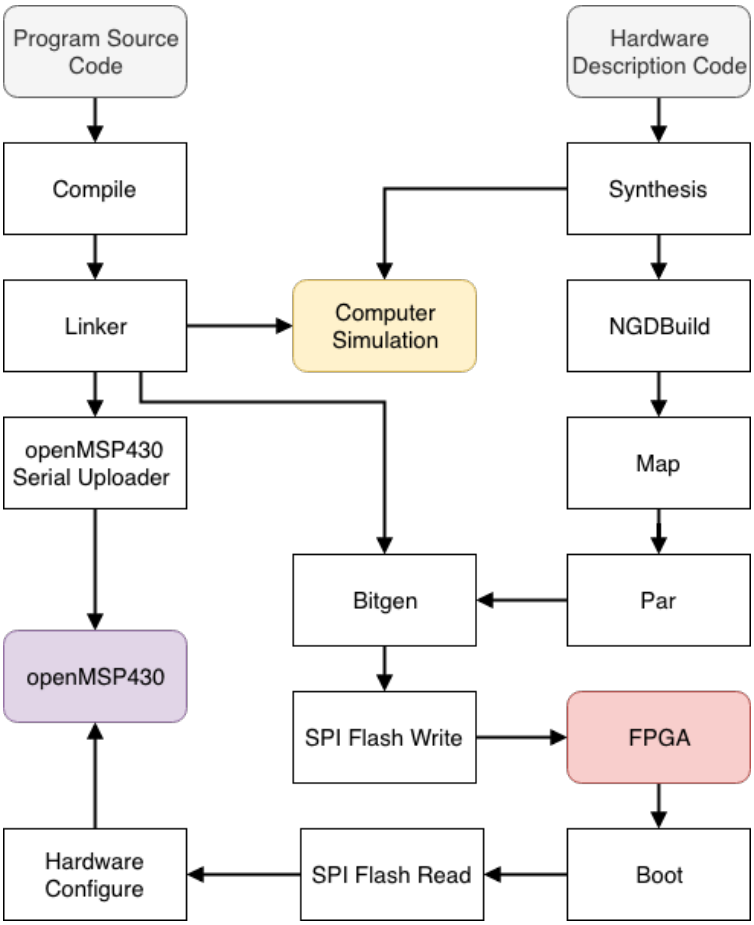

Figure 2.1: *Development building and load steps.*

Chapter 3

Implementing SMART

3.1 Premises

In the main article is presented few assertions to ake an informal proof of the SMART security. They are:

- A1: is impossible to forge the hash value. One significant difference from the main article is the fact that the hash is computed using special hardware and not software code. This hardware is designed to receive a data chunk and digest it to make the hash value. There is a guarantee on a hardware level to make impossible to recover the data sent to the hash computation.
- A2: the device with SMART will not suffer any hardware attack, only software.
- A3: only the SMART code can access the SMART key.
- A4: only the SMART code can change itself. This assertion is different from the original article but will provide additional features.
- A5: SMART code can only be called from the first memory address.
- A6: if the SMART code moves the instruction pointer from outside its region, it is impossible to return to the previous code. This assertion is a little different from the original but provides the same guarantees.
- A7: all interrupts are turned off during the SMART code execution.
- A8: after the SMART code execution the SMART key cannot be recovered. The hash hardware and software clean in all memory used in the SMART code guarantee this.
- A9: the SMART code compute the hash correctly after receiving a challenge.
- A10: after a reset, erase all memory data segment. This procedure prevents the leak of any information remaining in the memory.
- A11: none information from the SMART key can be a leak.

3.2 Implementation Details

One of the main objectives of this work is to implement the SMART as described in the article. The authors made two implementations of SMART, one in a Atmel AVR microcontroller and other in a Texas Instruments MSP430. Both are described

3.2.1 MSP430

falar sobre ele, como a memoria Ã© inicializada.... mostrar organizaÃ§ao de memoria....

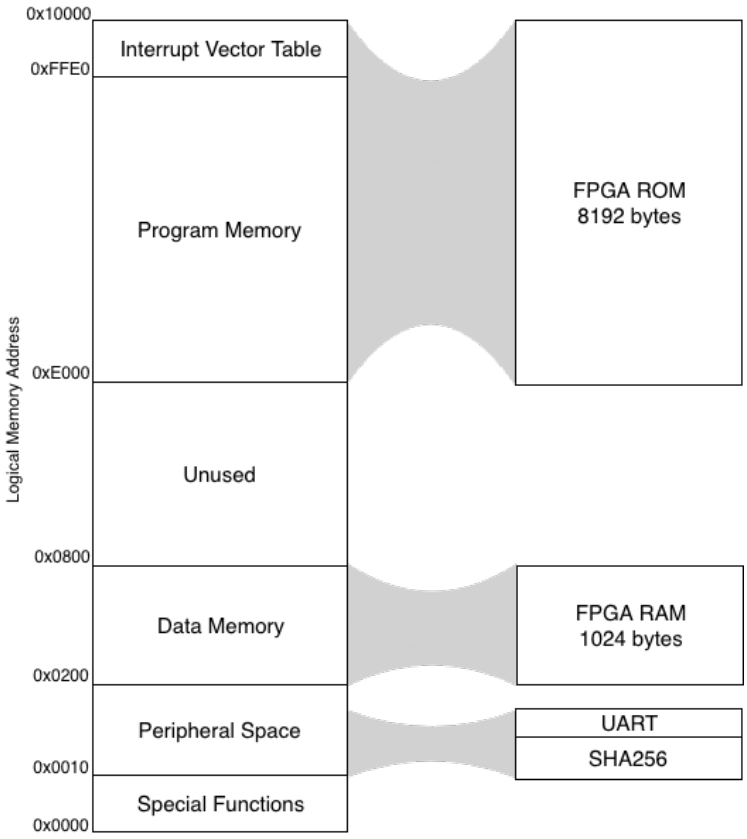

Figure 3.1: *a nice plot*

3.2.2 SPARTAN 6

3.2.3 Time Constants

One important thing when working with FPGA is the time constants. These values are essential for testing, and the dispose of the connections inside the FPGA.

For Verilog simulations, the time constants are set using a reserved macro named `timescale`. This macro receives two values: the time unit and time precision. The simulations delays and any time value use the time unit. And the time precision is used by the simulator to know how they can round the summation values. In the tests are used $1ns$ for time unit and $100ps$ for precision.

```
1 initial
    begin
3       CLK_100MHz = 1'b0;
        forever #5 CLK_100MHz <= ~CLK_100MHz; // 100 MHz
5   end
```

Listing 3.1: *Simulation clock signal generator.*

Code 3.1 is used in simulation to generate the clock signal. It changes the value of wire `CLK_100MHz` in intervals of $5ns$ (5 units of time), making an output signal of $100MHz$. Is chosen the frequency of $100MHz$ because the testing FPGA has a clock of this frequency.

These values are calculated using the frequency formula $F = \frac{1}{T}$, where $F = 100MHz$ is the desired frequency and T is the period. After solving this equation $T = 10^{-8}s$, this is equal to 10 units of time (10ns). The FPGA clock uses a duty cycle of 50%, that is that the clock will need to be 50% of their time active and the other part inactive (the signal will change to high to low or vice-versa in an interval of 5 units of time).

The communication with the simulator in testing or the computer in a real FPGA is made using the RS-232 standard. This standard is a serial protocol, the bits are sent one by time. A direct consequence is that the protocol use time constraints to send the information. In all test is used the 19200 bit/s baud rate, making each bit transmission time be $52100ns$.

To reduce the timing conflicts and problems inside the openMP430 [4] the microcontroller will receive a clock input of $20000MHz$. To change from $100000MHz$ to the desired input speed, a Digital Clock Manager (DCM) [1] will be used. The DCM is a dedicated hardware inside the FPGA for clock frequency conversion. To this module works it needs several configuration parameters, to simplify it a module named `clock.v` was created with all configurations inside it.

The microcontroller uses one hardware peripheral to interface with the RS-232 serial port. This hardware need to know the bit transmission time, in all software that uses serial has the `UART_BAUD = BAUD;` line. This line is responsible for setting a unique memory address to the correct bit transmission time.

3.2.4 SHA256

One improvement in this implementation compared to the original article has using a hardware SHA2 (Secure Hash Algorithm 2) to hash the memory. The original implementation uses a software SHA1. Different from SHA1, SHA2 is a family of hash function, in implementation use the SHA256 function.

The SHA256 has some improvements compared to the SHA1. The first one has the input size, SHA1 needs to receive only 160 bits to produce a hash, the SHA256 needs 256 bits. SHA1 was deprecated by NIST (National Institute of Standards and Technology) in 2011. There also some articles, like [10], to show how an attacker can forge two distinct PDF documents with the same SHA-1 hash.

It is important to notice that exist a new version of the secure hash algorithm, the SHA3. This version made some improvements and NIST advise to use it. There was the attempt to implement the SHA3 in the FPGA, but it exceeds the number of available LUTs in the FPGA, making the implementation impossible with the openMSP430 core. However, SHA256 is still considered segure, and it fit inside the FPGA.

A SHA3 implementation written by Joachim Strombergson [2] has used. A peripheral adaptor was built to communicate with the openMSP430 core. This adaptor is responsible for interfacing the SHA3 implementation with the peripherals pins in the microcontroller. The file `SMART/rtl/verilog/sha256/sha256per.v` contains this interface.

Another change is the use of a hardware hash function and not a software one. This change makes the hash timing faster and saves several bytes in ROM. However, it uses hardware approximately 2000 LUTs.

3.2.5 Linker Script

When building the binary to a microcontroller the compiler needs to know where place the code, data and other pieces of information in the memory. Linker script (or linker command file) is a particular file that describes to the compiler this attributes. Original the MSP430

mapped all memory using 16 bits addresses, making the maximum available space to be 65536 bytes.

However, the majority of microcontrollers compatible with the MSP430 architecture don't use all this memory.

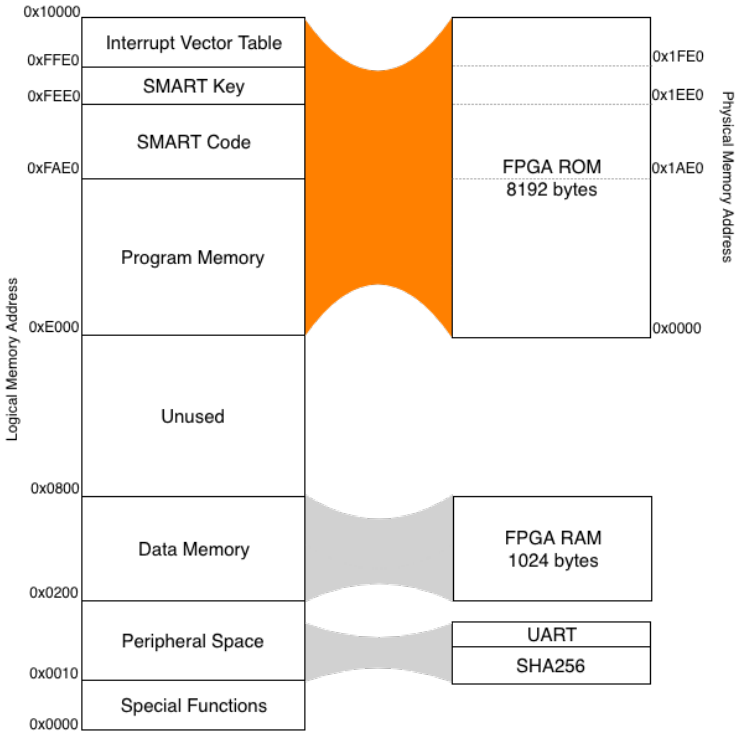

Figure 3.2: *a nice plot*

falar do original e da diferenÃ§a do que foi feito

3.2.6 MCAM

3.3 Tests

3.3.1 Continuous integration

A continuous integration system is used to prevent and identify any code error or change that affected the correct work of the microcontroller. A GitHub project is set to track the code changes and versioning the code. On every push, a server receives a notification, run all test in the project and build the FPGA files.

Jenkins[1] is used to build the continues integration. Other systems, like Travis or Gitlab CI, are not used because they have restrictions on the maximum size of their builds. To test and build the code the Xilinx ISE WebPACK Design Software is used, this software has approximately 6 GBytes. Jenkins makes possible install this software on the tester machine only one time and uses it when needed.

Are used two software to simulate the behavior of the hardware: the Icarus Verilog[2] (an open-source project) and the Xilinx ISE (a property software). For every test are tested using this two softwares, this decision is made because during the development are notice that some errors only occur in one of them.

[1] https://jenkins.io/
[2] http://iverilog.icarus.com/

3.4 Results

Chapter 4

Improving SMART

The opemMSP430 memory backbone is changed, allowing writing in the program memory. This change makes possible remote device updates, but also introduces several security breaches. However, the SMART implementation can identify if any breach occurs.

A system is built to control, verify and make a remote update using the SMART as the final example.

The article *ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices*[5] has the base of this system.

Chapter 5

Conclusion

Texto texto[1].

[1]Exemplo de referência para página Web: www.vision.ime.usp.br/~jmena/stuff/tese-exemplo

Appendix A

Módulo de Controle de Acesso à Memória

O objetivo desse apêndice é descrever o funcionamento do módulo o módulo de controle de acesso a memória (MCAM) implementado nesse trabalho.

Esse módulo permite que uma região da memória seja lida somente se uma função for executada. Ele verifica se o ponteiro de instrução é igual a um valor específico. Caso seja, habilita que uma região protegida da memória seja lida. Ao sair dessa função ele inibe o acesso à essa região. Caso haja a tentativa de acessar uma região sem autorização, o módulo reinicia o dispositivo.

A descrição de cada entrada e saída está presente na tabela a seguir:

Nome do Parâmetro	**Tipo**	**Tamanho (bits)**	**Descrição**
`in_safe_area`	Saída	1	Quando ativado (ligado) simboliza que o acesso à região da memória protegia está liberado. Ou seja, a função que terá acesso a essa região está sendo executada.
`reset`	Saída	1	Quando ativado o dispositivo tem que ser reiniciado.
`mem_dout`	Saída	16	Saída dos dados da memória.
`mem_addr`	Entrada		Endereço da memória à ser acessada. O seu tamanho pode ser configurada via parâmetros.
`mem_din`	Entrada	16	Entrada dos dados da memória.
`mclk`	Entrada	1	Relógio da memória.
`ins_addr`	Entrada	16	Endereço que o ponteiro de instruções está usando.
`disable_debug`	Entrada	1	Quando ativado, desabilita o módulo. Útil para depuração.

Além das entradas e saídas, é possível entrar com alguns parâmetros na confecção do módulo. A tabela abaixo mostra quais são esses parâmetros e suas funcionalidades.

Nome do Parâmetro	**Descrição**
SIZE_MEM_ADDR	Valor que representa o número de bits presente no endereço de memória (`mem_addr`) menos um.
LOW_SAFE	O menor endereço físico na memória da região à ser protegida.
HIGH_SAFE	O maior endereço físico na memória da região à ser protegida.
LOW_CODE	Onde começa a função que terá acesso a região protegida na memória. Endereço virtual.
HIGH_CODE	Onde termina a função que terá acesso a região protegida na memória. Endereço virtual.

Note que para o endereçamento das funções que tem acesso a parte protegida é utilizado o endereço virtual. A descrição do espaço de memória protegida utiliza o endereço físico. O endereço virtual é o utilizado na pelo programa e mantido pelo compilador. Já o endereço físico provém do virtual adaptado para o tipo de memória conectado ao dispositivo. Aqui utilizamos dois tipos de endereçamento diferente, pois o ponteiro de instruções utiliza o endereço virtual. Em contrapartida, o acesso de memória utiliza o físico nativamente. Uma descrição mais detalhada sobre as diferenças de endereçamento e como realizar a conversão de um para outro pode ser vista na página do openMP430 [4].

No artigo é citado que para criação do controle de acesso a memória é necessária poucas modificações. Essa afirmação ;e verídica, no entanto as modificações devem ser feitas de forma precisa. Em nenhum momento no artigo é citado que o controle da memória deve ser sincronizado com o relógio de acesso a mesma. Apesar dessa premissa ser óbvia ela é relevante, pois se somente observamos o endereço da memória, sem levar em conta o relógio, haverá momentos em que o dispositivo será reiniciado sem a necessidade.

A implementação do módulo pode ser separada em duas partes principais. A primeira verifica se o ponteiro de instruções entrou corretamente no bloco de código que tem direito ao acesso à região protegida. Essa verificação é efetuada comprando o ponteiro de instruções do microcontrolador com os parâmetros do módulo. Caso o valor do ponteiro `ins_addr` seja igual a LOW_CODE, temos que valor dentro do hardware se torna positivo, habilitando o acesso. Quando o `ins_addr` $\geq$ HIGH_CODE ou `ins_addr` $\leq$ LOW_CODE-1, esse valor se torna 0, impedindo o acesso.

A segunda parte é responsável por verificar se o endereço de memória está tentando ler uma região protegida. Ela verifica se LOW_CODE $\leq$ `mem_addr` $\leq$ HIGH_CODE. Caso essa condição seja verdade, mas o dispositivo não tenha executado a função da primeira parte, ele é reiniciado.

A implementação completa desse módulo pode ser vista no repositório Github desse trabalho [1].

[1] https://github.com/capellaresumo/MAC0499/blob/master/SMART/rtl/verilog/mcam.v

Bibliography

[1] *Spartan-6 FPGA Clocking Resources*, 2018. https://www.xilinx.com/support/documentation/user_guides/ug382.pdf [Accessed: September 2018]. 7

[2] *Hardware implementation of the SHA-256 cryptographic hash function*, 2018. https://github.com/secworks/sha256 [Accessed: September 2018]. 7

[3] *LegUp High-Level Synthesis*, 2018. http://legup.eecg.utoronto.ca/ [Accessed: September 2018]. 3

[4] *openMSP430*, 2018. https://opencores.org/project/openmsp430 [Accessed: August 2018]. 7, 13

[5] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. Assured: Architecture for secure software update of realistic embedded devices, 07 2018. 10

[6] K. Eldefrawy, D. Perito, and G. Tsudik. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. 01 2012. 3

[7] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. pages 1–6, 01 2014. 3

[8] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 10:1–10:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592824. URL http://doi.acm.org/10.1145/2592798.2592824. 3

[9] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3):7:1–7:33, July 2017. ISSN 2471-2566. doi: 10.1145/3079763. URL http://doi.acm.org/10.1145/3079763. 3

[10] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full sha-1, 07 2017. 7