PAuth ABI Extension to ELF for the Arm® 64-bit Architecture (AArch64)

2023Q3

Date of Issue: 6th October 2023

1 Preamble

1.1 Abstract

This document describes the PAuth ABI Extensions to ELF for the Arm 64-bit architecture (AArch64). A high-level language may use these extensions to make wider use of pointer authentication than is possible within the base standard.

1.2 Keywords

ELF, AArch64 ELF, Pointer Authentication

1.3 Latest release and defects report

Please check Application Binary Interface for the Arm® Architecture for the latest release of this document.

Please report defects in this specification to the issue tracker page on GitHub.

1.4 Licence

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Grant of Patent License. Subject to the terms and conditions of this license (both the Public License and this Patent License), each Licensor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Licensed Material, where such license applies only to those patent claims licensable by such Licensor that are necessarily infringed by their contribution(s) alone or by combination of their contribution(s) with the Licensed Material to which such contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Licensed Material or a contribution incorporated within the Licensed Material constitutes direct or contributory patent infringement, then any licenses granted to You under this license for that Licensed Material shall terminate as of the date such litigation is filed.

1.5 About the license

As identified more fully in the Licence section, this project is licensed under CC-BY-SA-4.0 along with an additional patent license. The language in the additional patent license is largely identical to that in Apache-2.0 (specifically, Section 3 of Apache-2.0 as reflected at https://www.apache.org/licenses/LICENSE-2.0) with two exceptions.

First, several changes were made related to the defined terms so as to reflect the fact that such defined terms need to align with the terminology in CC-BY-SA-4.0 rather than Apache-2.0 (e.g., changing “Work” to “Licensed Material”).

Second, the defensive termination clause was changed such that the scope of defensive termination applies to “any licenses granted to You” (rather than “any patent licenses granted to You”). This change is intended to help maintain a healthy ecosystem by providing additional protection to the community against patent litigation claims.

1.6 Contributions

Contributions to this project are licensed under an inbound=outbound model such that any such contributions are licensed by the contributor under the same terms as those in the Licence section.

1.7 Trademark notice

The text of and illustrations in this document are licensed by Arm under a Creative Commons Attribution–Share Alike 4.0 International license ("CC-BY-SA-4.0”), with an additional clause on patents. The Arm trademarks featured here are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Please visit https://www.arm.com/company/policies/trademarks for more information about Arm’s trademarks.

1.8 Copyright

Copyright (c) 2020-2023, Arm Limited and its affiliates. All rights reserved.

Contents

2 About this document

2.1 Change Control

2.1.1 Current Status and Anticipated Changes

The following support level definitions are used by the Arm ABI specifications:

Release

Arm considers this specification to have enough implementations, which have received sufficient testing, to verify that it is correct. The details of these criteria are dependent on the scale and complexity of the change over previous versions: small, simple changes might only require one implementation, but more complex changes require multiple independent implementations, which have been rigorously tested for cross-compatibility. Arm anticipates that future changes to this specification will be limited to typographical corrections, clarifications and compatible extensions.

Beta

Arm considers this specification to be complete, but existing implementations do not meet the requirements for confidence in its release quality. Arm may need to make incompatible changes if issues emerge from its implementation.

Alpha

The content of this specification is a draft, and Arm considers the likelihood of future incompatible changes to be significant.

This document is at **Alpha** release quality.

2.1.2 Change history

If there is no entry in the change history table for a release, there are no changes to the content of the document for that release.

Issue	Date	Change
0.1	21st September 2020	Alpha draft release.
0.2	7th October 2020	Restructure after initial review comments. Move GOT relocations with custom signing schema to appendix along with commentary. Specify AUTH variant dynamic relocations and tighten up descriptions of existing encoding and relocations. Add description of SHT_AUTH_RELR.
0.3	22nd October 2020	Delete the appendix giving an encoding for GOT generating relocations using the relocation addend to carry the signing schema. Move the remaining GOT signing text to an optional appendix Describe the default dlsym signing schema and add .symauth and .dynauth as an appendix. Provide details for the ELF marking scheme. description of SHT_AUTH_RELR.
2021Q3	1st November 2021	Clarify the concepts of a default and alternate PAuth elf signing schemes.
2023Q3	6th October 2023	Update tags in Dynamic Section to avoid conflict with DT_AARCH64_VARIANT_PCS.

2.2 References

This document refers to, or is referred to by, the following documents.

Ref	URL or other reference	Title
ARMARM	DDI 0487	Arm Architecture Reference Manual Armv8 for Armv8-A architecture profile
AAELF64	IHI 0056	ELF for the Arm 64-bit Architecture
ARM64E		Pointer Authentication
CPPABI64	IHI 0059	C++ ABI for the Arm 64-bit Architecture
LSB		Linux Standards Base
SCO-ELF	http://www.sco.com/developers/gabi/	System V Application Binary Interface – DRAFT
TLSDESC	http://www.fsfla.org/~lxoliva/writeups/TLS/paper-lk2006.pdf	TLS Descriptors for Arm. Original proposal document
GABI_SHT_RELR	ELF GABI Google Groups	Proposal for a new section type SHT_RELR
LINUX_ABI	https://github.com/hjl-tools/linux-abi/wiki	Linux Extensions to gABI

2.3 Terms and Abbreviations

The ABI for the Arm 64-bit Architecture uses the following terms and abbreviations.

A32

The instruction set named Arm in the Armv7 architecture; A32 uses 32-bit fixed-length instructions.

A64

The instruction set available when in AArch64 state.

AAPCS64

Procedure Call Standard for the Arm 64-bit Architecture (AArch64)

AArch32

The 32-bit general-purpose register width state of the Armv8 architecture, broadly compatible with the Armv7-A architecture.

AArch64

The 64-bit general-purpose register width state of the Armv8 architecture.

ABI

Application Binary Interface:

1. The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the *Linux ABI for the Arm Architecture*.
2. A particular aspect of the specifications to which independently produced relocatable files must conform in order to be statically linkable and executable. For example, the CPPABI64, AAELF64, ...

Arm-based

... based on the Arm architecture ...

Floating point

Depending on context floating point means or qualifies: (a) floating-point arithmetic conforming to IEEE 754 2008; (b) the Armv8 floating point instruction set; (c) the register set shared by (b) and the Armv8 SIMD instruction set.

Q-o-I

Quality of Implementation – a quality, behavior, functionality, or mechanism not required by this standard, but which might be provided by systems conforming to it. Q-o-I is often used to describe the toolchain-specific means by which a standard requirement is met.

SIMD

Single Instruction Multiple Data – A term denoting or qualifying: (a) processing several data items in parallel under the control of one instruction; (b) the Arm v8 SIMD instruction set: (c) the register set shared by (b) and the Armv8 floating point instruction set.

SIMD and floating point

The Arm architecture's SIMD and Floating Point architecture comprising the floating point instruction set, the SIMD instruction set and the register set shared by them.

SVE

The Arm architecture's Scalable Vector Extension.

T32

The instruction set named Thumb in the Armv7 architecture; T32 uses 16-bit and 32-bit instructions.

VG

The number of 64-bit “vector granules” in an SVE vector; in other words, the number of bits in an SVE vector register divided by 64.

ILP32

SysV-like data model where int, long int and pointer are 32-bit

LP64

SysV-like data model where int is 32-bit, but long int and pointer are 64-bit.

LLP64

Windows-like data model where int and long int are 32-bit, but long long int and pointer are 64-bit.

This document uses the following terms and abbreviations.

Link-unit

An executable or shared library

PAuth ABI

The pointer authentication ABI that this document forms a part of.

PAUTHELF64

An abbreviation for this document.

RELRO

Part of an ELF file that can be mapped read-only after relocation. In an executable/shared-library it is described by a program header with type PT_GNU_RELRO.

Signing Schema

The set of rules that determine how a pointer is signed. In ARMARM terminology the rules will evaluate to a key and a modifier that can be used in a signing or authorizing operation.

Default signing schema

A default signing schema for a pointer is determined by the context. The signing schema will not be encoded in the ELF file.

3 Scope

This document is a set of extensions to ELF for the Arm 64-bit architecture (AAELF64) describing how PAuth ABI information is encoded in the ELF file. As an alpha document all details in this document are subject to change.

4 Platform Standards

As is the case with the AAELF64, we expect that each operating system that adopts components of this ABI specification will specify additional requirements and constraints that must be met by application code in binary form and the code-generation tools that generate such code. This document will present recommendations for a SysVr4 like operating system such as Linux.

5 Introduction

The Armv8.3-A architecture introduced a pointer authentication feature that permits a pointer to be cryptographically signed and authenticated. A subset of the new instructions were added in the HINT space to take advantage of a limited form of pointer authentication that maintained backwards compatibility with software written without assuming Armv8.3-A capabilities. If use of all of the PAuth instructions is permitted then more pointers can be protected at the expense of requiring Armv8.3-A and potential incompatibility with objects not using the PAuth ABI.

5.1 Design Goals

The goals of the final PAUTHELF64 document are to:

- Provide primitives that can be used to support different language and platform choices for a PAuth ABI, including the minimal bare-metal platform.
- Provide a means to reason about compatibility of ELF files at both the relocatable and executable/shared-library level.

The goals of the initial draft of the PAUTHELF64 document are to:

- Enable experimentation to find out the most useful encodings and options.
- Provide rationale for design choices.

5.2 General Principles

- Signed pointers can only be created at run-time.

5.3 General Restrictions

- PAUTHELF64 does not support the R_AARCH64_COPY relocation for signed pointers. Non-position independent code that imports signed pointers from shared libraries must use an alternative code-sequence that does not require the static linker to use COPY relocations. A simple way to avoid COPY relocations is to access imported signed pointers via the GOT.
- PAUTHELF64 only supports the descriptor based TLS (TLSDESC).

The Rationale behind the requirement to avoid copy relocations is that the static linker creates the storage that the copy is placed; which adds more complication in the form of communicating a signing schema than avoiding the copy relocation. The descriptor based TLS has been chosen as the most common implementation choice for AArch64.

6 Platform Decisions

PAUTHELF64 supports deployment of pointer authentication in a wide variety of environments including a bare-metal environment without a memory management unit. Platforms may not need to implement all of this ABI by placing additional platform specific restrictions. For example if the platform does not support lazy binding and both the GOT and PLT GOT are RELRO then there is no need to implement support for AUTH variant dynamic relocations. Optional parts of the ABI have been broken out into appendices.

6.1 RELRO GOT

The GOT is a linker generated table of pointers, where each entry is created as a result of a GOT-creating relocation from a relocatable object. The GOT is normally split into two subsets, the GOT and the PLT GOT. With the PLT GOT made up of code pointers that are only accessed by linker generated PLT sequences. As the PLT GOT is not accessed directly by code in relocatable objects, this will be covered in a separate section of the document.

The pointers in the GOT may include pointers to code so there is a question of whether these pointers should be signed and if so, how should they be signed?

The value of many of the pointers in the GOT is not known at static link time. These GOT entries will have a dynamic relocation that a dynamic linker will resolve at program load time. Once the GOT has been relocated, it can be re-mapped as read-only (RELRO). If the GOT is RELRO then the GOT does not need to be signed.

As a majority of platforms support RELRO and assuming a RELRO GOT simplifies the ABI, this document will assume an unsigned GOT. An optional appendix Appendix thoughts on encoding a signing schema describes how a GOT can be signed.

6.2 PLT GOT signing

The PLT is a table of trampolines used to indirect function calls through a function pointer. The PLT GOT is a subset of the GOT that is used exclusively by the PLT.

When lazy binding is disabled, all relocations are resolved at load time and the PLT GOT can be made RELRO like the GOT. In this case the PLT GOT does not need to be signed.

When lazy binding is enabled the initial contents of each entry in the PLT GOT points to the first entry in the PLT. The first entry in the PLT reserved for the lazy resolver. When a function is called via its PLT entry, control is transferred to the lazy resolver which finds the destination address and writes it back to the PLT GOT. As the lazy resolver needs to write to PLT GOT, it requires the PLT GOT to be writeable throughout the lifetime of the program. Writeable pointers in the PLT GOT can be signed to protect against an attacker modifying the PLT GOT.

If the PLT GOT is signed, the dynamic linker must sign all entries in the table during program loading and the static linker must generate PLT trampolines that authenticate pointers loaded from the PLT GOT.

The choice of whether to sign the PLT GOT is made at static link time. The decision to sign the PLT GOT is independent of the decision to sign the GOT.

6.2.1 PLT GOT signing schema

The PAuth ABI reuses the signing schema from the existing ABI. This uses the IA key with the address of the PLT GOT entry as the modifier. This can be implemented using instructions that are encoded in the hint space.

Example code for the PLT sequence generated by the static linker.

```
adrp x16, Page(&(.plt.got[n]))
ldr  x17, [x16, Offset(&(.plt.got[n]))]
add  x16, x16, Offset(&(.plt.got[n]))
autia1716
br   x17
```

If instructions not encoded in the hint space can be used, it is possible to combine the `autia1716` and `br x17` into a single instruction `braa x17, x16`

6.2.2 Recording a signed PLT GOT in the ELF file

The dynamic tag `DT_AARCH64_PAC_PLT` must be set if the PLT GOT is signed. When this dynamic tag is present, a dynamic loader must sign the result of a `R_AARCH64_JUMP_SLOT` using the signing schema above. PAUTHELF64 does not define an additional `R_AARCH64_AUTH_JUMP_SLOT` relocation as the static linker will only generate one type of PLT entry in a link-unit.

7 Section Types

The PAuth ABI adds an additional Processor-specific section type

ELF Section Types

Name	Value	Comment
`SHT_AARCH64_AUTH_RELR`	`0x70000004`	Section type for compressed signed relative relocations

The value is in the AArch64 Processor-specific range. The value is subject to change if there is a clash with any section types added by AAELF64.

8 Static Relocations

As the PAuth ABI is Alpha, relocation codes are in the vendor experiment space of 0xE000 to 0xEFFF. These are guaranteed not to clash with any relocation type defined by Arm in AAELF64. New permanent relocation codes will be issued in AAELF64 when the PAuth ABI comes out of Alpha.

8.1 Encoding of authenticated pointer

This ABI requires the creation of signed pointers at program start-up by the run-time environment. The signing schema to be used by the run-time environment is encoded in the place to be relocated.

8.1.1 Encoding the signing schema

In the descriptions below, the `place` is the operation `P` in AAELF64 relocation descriptions. It is derived from the r_offset field of the relocation.

The top 32-bits of the contents of the place is used to encode the signing schema for both static and dynamic relocations. This permits platforms using relocation compression or SHT_REL dynamic relocations to encode relocation addends in the bottom 32-bits. Given that the maximum size of link-units using the small code-model is 4 gigabytes this should be sufficient.

Signing schema encoding

63	62	61:60	59:48	47:32	31:0
address diversity	reserved	key	reserved	discriminator	reserved for addend

- `address diversity` is a single bit that when set, denotes that the pointer has address diversity. The place (relocation target address) will be blended with the discriminator value.
- `key` determines the key to be used. Armv8.3-A specifies 5 keys, 4 of which can be used by PAUTHELF64. The generic key APGA is not repesented at the ELF ABI level.

Key encoding

key name	field
`APIA`	0b00
`APIB`	0b01
`APDA`	0b10
`APDB`	0b11

- `discriminator` is a 16-bit unsigned integer that after an optional blending (address diversity) forms the `modifier` for the sign and authenticate instructions.
- `reserved for addend` is used in `SHT_AUTH_RELR` or `SHT_REL` relocation implementations where the relocation addend is written to the contents of the place. It must be set to 0 if not used for an addend.
- `reserved` are bits reserved for future expansion. These bits must be set to 0 by a producer. A consumer must not assume that reserved bits are set to 0.

For a relocation that involves signing a pointer, if the target symbol is an undefined weak reference, the result of the relocation is 0 (nullptr) regardless of the signing schema.

The computation to form the `modifier` is the same as ARM64E. `Place` is the relocation target address.

- If `address diversity` is set and the `discriminator` is 0 then `modifier` = `Place`
- If `address diversity` is set and the `discriminator` is not 0 then `modifier[63:48]` = `discriminator` and `modifier[47:0]` = `Place`
- If `address diversity` is not set then `modifier` = `discriminator` zero-extended to 64-bits.

8.2 Relocation Operations

- `PAUTH(S+A)` is an instruction for the run-time environment to create a signed pointer. The static linker writes the encoded signing schema into the contents of the place being relocated, and emits a dynamic relocation to instruct the run-time to create the signed pointer. When static linking, the dynamic relocation may be replaced by a toolchain-specific mechanism.
- `SCHEMA(\*P)` represents the dynamic linker reading the signing schema from the contents of the place `P`.
- `SIGN(value, schema)` represents the dynamic linker signing value with schema.

8.3 Static Data relocations

PAuth Data relocations

ELF 64 Code	Name	Operation	Comment
0xE100	R_AARCH64_AUTH_ABS64	`PAUTH(S+A)`	Signing schema encoded in the contents of the place

In the static context, this is the equivalent of the arm64e `ARM64_RELOC_AUTHENTICATED` relocation. `R_AARCH64_AUTH_ABS64` can also be used as a dynamic relocation with the same ELF 64 Code.

9 AUTH variant Dynamic Relocations

The dynamic relocations required for the PAuth ABI are built on the existing dynamic relocations, for example `R_AARCH64_AUTH_RELATIVE` is the PAuth ABI equivalent of `R_AARCH64_RELATIVE`. The underlying calculation performed by the dynamic linker is the same, the only difference is that the resulting pointer is signed. The dynamic linker reads the signing schema from the contents of the place of the dynamic relocation.

Dynamic relocations

Relocation code	Name	Operation
0xE100	R_AARCH64_AUTH_ABS64	SIGN(S + A, SCHEMA(*P))
0xE200	R_AARCH64_AUTH_RELATIVE	SIGN(DELTA(S) + A, SCHEMA(*P))

10 Dynamic Section

The PAuth ABI adds the following processor-specific dynamic array tags.

Additional AArch64 specific dynamic array tags

Name	Value	d_un	Executable	Shared Object
DT_AARCH64_AUTH_RELRSZ	0x70000011	d_val	optional	optional
DT_AARCH64_AUTH_RELR	0x70000012	d_ptr	optional	optional
DT_AARCH64_AUTH_RELRENT	0x70000013	d_val	optional	optional

Description:

- `DT_AARCH64_AUTH_RELRSZ` This element holds the total size in bytes, of the `DT_AARCH64_AUTH_RELR` relocation table.
- `DT_AARCH64_AUTH_RELR` The address of an `SHT_AARCH64_AUTH_RELR` relocation table. This element requires the `DT_AARCH64_AUTH_RELRSZ` and `DT_AARCH64_AUTH_RELRENT` elements also be present. During dynamic linking, a `DT_AARCH64_AUTH_RELR` element is processed before any `DT_REL` or `DT_RELA` elements in the same object file.
- `DT_AARCH64_AUTH_RELRENT` This element holds the size in bytes of a `DT_AARCH64_RELR` relocation entry.

11 Relocation Compression

The SHT_RELR section type as defined in GABI_SHT_RELR, when present in an AArch64 ELF file, encodes `R_AARCH64_RELATIVE` relocations in a more compact form. To encode `R_AARCH64_AUTH_RELATIVE` using the same encoding, a new ELF section type `SHT_AARCH64_AUTH_RELR` is added, alongside the dynamic tags `DT_AARCH64_AUTH_RELR`, `DT_AARCH64_AUTH_RELRENT`, and `DT_AARCH64_AUTH_RELRSZ`.

The format of the `SHT_AARCH64_AUTH_RELR` section is identical to `SHT_RELR`, the only difference is that all relocations are of type `R_AARCH64_AUTH_RELATIVE`. A link-unit may contain both `SHT_RELR` and `SHT_AARCH64_AUTH_RELR` sections.

12 Static Linking

The static linker cannot create signed pointers, just as it cannot run constructors for static variables, but the start-up code that runs before the program (in C/C++, before `main()`) can. The static linker must communicate the details of how to create the signed pointers by embedding the information in the ELF file. The format of the information must be defined by the platform ABI, as it is a contract between the static linker and the runtime. One simple method of encoding the information is to emit a dynamic relocation section as if dynamic linking, with linker-defined symbols denoting the base and limit of the section. The runtime can resolve the dynamic relocations to create the signed pointers. More compact encodings are possible.

13 Run-time dynamic linking

On many platforms, programs can load shared libraries at run-time via `dlopen` and access symbols in that library via `dlsym` or `dlvsym`. Some or all of these pointers may be signed. The signing schema for these functions is a platform decision that the compiled code and implementation of `dlsym` agree on.

The PAuth ABI uses a simple default signing schema. If the symbol found by `dlsym` has type `STT_FUNC`, the address to be returned is signed with the `IA` key with a 0 modifier. Otherwise the address is not signed.

An optional extension that communicates the signing schema for a symbol to the dynamic linker can be found in Appendix extension to recording signing schema for dlsym.

14 ELF Marking

ELF files must be marked to allow toolchains and platforms to reason about compatibility. The high-level language mapping of source language to signing schema is expected to evolve over time. Even if the low-level ELF extensions remain constant, a change to the high-level language mapping may result in incompatible ELF files.

This document defines a default ELF marking schema and a base compatibility model. Platforms may define their own ELF marking and compatibility model that replace or extend the default ones. Appendix Alternative ELF Marking Using GNU Program Properties defines an alternative marking schema for platforms that support the `.note.gnu.property` section.

14.1 Default Marking Schema

A new section named `.note.AARCH64-PAUTH-ABI-tag` of type `SHT_NOTE` is defined. This section is structured as a note section as documented in SCO-ELF, and its attribute flag `SHF_ALLOC` must be set.

The name field (`namesz` / `name`) contains the string "ARM". The type field shall be 1, and the `descsz` field must be at least 16. The first 16 bytes of the description must contain 2 64-bit words, with the first 64-bit word being a platform identifier, and the second 64-bit word being a version number for the ABI for the platform identified for the first word. When `descsz` is larger than 16 the remainder of the contents of desc are defined by the (platform id, version number).

The following values of the platform id are reserved:

Platform	Hex value
Invalid	0x0
Baremetal	0x1

The version number in `.note.AARCH64-PAUTH-ABI-tag` is not directly related to the version number of this document. It is controlled by the object-producer based on the signing schema that has been used for pointers.

If a file contains a section named `.note.AARCH64-PAUTH-ABI-tag`, it must observe the entirety of the rules in this default marking schema. Generating such section with a platform-specific schema is forbidden.

14.2 Base Compatibility Model

A per-ELF file marking scheme permits a coarse way of reasoning about compatibility.

- The absence of a `.note.AARCH64-PAUTH-ABI-tag` section means no information on how pointers are signed is available for this ELF file.
- The presence of a `.note.AARCH64-PAUTH-ABI-tag` means that, if the file contains pointers, they were signed in a compatible way with the default signing rules for tuple (platform id, version number).
- The result of a successful combination of `.note.AArch64-PAUTH-ABI-tag` sections is a single `.note.AArch64-PAUTH-ABI-tag` section containing the (platform id, version number) tuple. The result of an unsuccesful combination must be either a single `.note.AArch64-PAUTH-ABI-tag` section containing a platform id with value Invalid, or no `.note.AArch64-PAUTH-ABI-tag` section written to the output file.
- The static linker may fault the combination of relocatable objects that contain `.note.AARCH64-PAUTH-ABI-tag` sections with incompatible (platform id, version number) tuples.
- A dynamic loader may disable pointer authentication, or fault the program with an error message, in case it encounters an incompatible ELF file. A file is incompatible with the loader in any of the following cases:
  - If section `.note.AARCH64-PAUTH-ABI-tag` is missing.
  - If the (platform id, version number) tuple is not recognized.
  - If the platform id is Invalid.

The combination of relocatable objects with `.note.AARCH64-PAUTH-ABI-tag` and relocatable objects without a `.note.AARCH64-PAUTH-ABI-tag` is not defined by this ABI.

Platforms may replace the base compatibility model with a platform-specific model.

15 Appendix extension to recording signing schema for dlsym

With additional per-symbol information encoded in the ELF file, a dynamic linker can look up the signing schema to use for `dlsym` or `dlvsym` instead of using the default signing schema.

15.1 .symauth and .dynauth sections

The pointer authentication information for global symbols is stored in a section named `.symauth` with type `SHT_AARCH64_AUTH_SYM`, it is associated with a symbol table section in a similar way to .symtab_shndx. The following table describes the `SHT_AARCH64_AUTH_SYM` symbol type:

`.symauth` and `.dynauth` ELF Section Type

Name	Value	Comment
SHT_AARCH64_AUTH_SYM	0x70000005	Section type for symbol signing information

A section of type `SHT_AARCH64_AUTH_SYM` is an array of `Elf32_Word` values. Each value corresponds to a non-local symbol table entry in the symbol table and appear in the same order as those entries. All local symbols in the symbol table precede global symbols so the index in `.symauth` of a global symbol with index `I` in the symbol table is `I - Index of first non-local symbol`. Each table entry is specified as follows:

.symauth and .dynauth entry encoding

31	30	30-19	18-17	16	15-0
sign	set	reserved	key	0	discriminator

- `key` same as in Encoding the signing schema.
- `discriminator` same as in Encoding the signing schema.
- `reserved` same as in Encoding the signing schema.
- `sign` indicates whether the address of the symbol should be signed when its addres is taken by `dlsym`.
- `set` indicates whether an assembly directive was used to set the signing schema. This may be used by the linker to detect cases where a directive was required but was not present.

There is no `address diversity` field as this has no meaning for symbols returned by `dlsym`.

For ELF shared libraries and executables that support dynamic linking the static linker creates a `SHT_AARCH64_AUTH_SYM` section with name `.dynauth`. This section is associated with the dynamic symbol table. If the `.dynauth` section is present an additional dynamic tag DT_AARCH64_AUTH_SYM is added.

.dynauth ELF dynamic tag

Name	Value	d_un	Executable	Shared Object
DT_AARCH64_AUTH_SYM	0x70000008	d_ptr	optional	optional

16 Appendix Signed GOT

If the program (-z norelo) or platform does not support RELRO the GOT will be writeable for the lifetime of the program. There is scope for some or all of the pointers in the GOT to protect against an attacker modifying the GOT.

If a pointer in the GOT is to be signed then the dynamic linker must sign the pointer at load time, and the code that loads the pointer from the GOT must authenticate it using the same signing schema.

PAUTHELF64 describes a default signing schema for GOT entries and AUTH variant GOT-generating relocations that can be used to create signed pointers in the GOT.

16.1 Default signing schema

Signed GOT entries use the IA key for symbols of type STT_FUNC and the DA key for all other symbol types, with the address of the GOT entry as the modifier. The static linker must encode the signing schema into the GOT slot. AUTH variant dynamic relocations must be used for signed GOT entries.

Example Code to access a signed GOT entry

```
adrp x8, :got_auth: symbol
add x8, x8, :got_auth_lo12: symbol
ldr x0, [x8]
// Authenticate to get unsigned pointer
autia x0, x8
```

In the example the :got_auth: and :got_auth_lo12: operators result in AUTH variant GOT generating relocations being used.

16.2 AUTH variant GOT Generating Relocations

ENCD(value) is the encoding of the signing schema into the GOT slot using the IA key for symbols of type STT_FUNC and the DA key for all other symbol types. The address of the GOT slot G is used as a modifer.

The GOT entries must be relocated by AUTH variant dynamic relocations.

PAuth GOT generating relocations

ELF 64 Code	Name	Operation	Comment
0x8110	R_AARCH64_AUTH_MOVW_GOTOFF_G0	G(ENCD(GDAT(S + A))) - GOT	Set a MOV[NZ] immediate field to bits [15:0] of X (see notes below)
0x8111	R_AARCH64_AUTH_MOVW_GOTOFF_G0_NC	G(ENCD(GDAT(S + A))) - GOT	Set a MOV[NZ] immediate field to bits [15:0] of X (see notes below)
0x8112	R_AARCH64_AUTH_MOVW_GOTOFF_G1	G(ENCD(GDAT(S + A))) - GOT	Set a MOV[NZ] immediate field to bits [31:16] of X (see notes below)

ELF 64 Code	Name	Operation	Comment
0x8113	R_AARCH64_AUTH_MOVW_GOTOFF_G1_NC	G(ENCD(GDAT(S + A))) - GOT	Set a MOV[NZ] immediate field to bits [31:16] of X (see notes below)
0x8114	R_AARCH64_AUTH_MOVW_GOTOFF_G2	G(ENCD(GDAT(S + A))) - GOT	Set a MOV[NZ] immediate field to bits [47:32] of X (see notes below)
0x8115	R_AARCH64_AUTH_MOVW_GOTOFF_G2_NC	G(ENCD(GDAT(S + A))) - GOT	Set a MOV[NZ] immediate field to bits [47:32] of X (see notes below)
0x8116	R_AARCH64_AUTH_MOVW_GOTOFF_G3	G(ENCD(GDAT(S + A))) - GOT	Set a MOV[NZ] immediate field to bits [63:48] of X (see notes below)
0x8117	R_AARCH64_AUTH_GOT_LD_PREL19	G(ENCD(GDAT(S + A))) - P	Set a load-literal immediate field to bits [20:2] of X; check $–2^{20}$ <= X < 2^{20}
0x8118	R_AARCH64_AUTH_LD64_GOTOFF_LO15	G(ENCD(GDAT(S + A))) - GOT	Set the immediate value of an ADRP to bits [32:12] of X; check that $–2^{32}$ <= X < 2^{32}
0x8119	R_AARCH64_AUTH_ADR_GOT_PAGE	G(ENCD(GDAT(S + A))) - Page(P)	Set the immediate value of an ADRP to bits [32:12] of X; check that $–2^{32}$ <= X < 2^{32}
0x811A	R_AARCH64_AUTH_GOT_LO12_NC	G(ENCD(GDAT(S + A)))	Set the LD/ST immediate field to bits [11:3] of X. No overflow check; check that X&7 = 0
0x811B	R_AARCH64_AUTH_LD64_GOTPAGE_LO15	G(ENCD(GDAT(S + A))) - Page(GOT)	Set the LD/ST immediate field to bits [14:3] of X; check that 0 <= X < 2^{15}
0x811C	RAARCH64_AUTH_GOT_ADD_LO12_NC	G(ENCD(GDAT(S + A)))	Set an ADD immediate value to bits [11:0] of X. No overflow check.

17 Additional AUTH variant Dynamic Relocations for Signed GOT

The dynamic relocations required for the PAuth ABI are built on the existing dynamic relocations, for example `R_AARCH64_AUTH_RELATIVE` is the PAuth ABI equivalent of `R_AARCH64_RELATIVE`. The underlying calculation performed by the dynamic linker is the same, the only difference is that the resulting pointer is signed. The dynamic linker reads the signing schema from the contents of the place of the dynamic relocation.

Additional AUTH Dynamic relocations

Relocation code	Name	Operation
0xE201	R_AARCH64_AUTH_GLOB_DAT	SIGN((S + A), SCHEMA(*P))
0xE202	R_AARCH64_AUTH_TLSDESC	SIGN(TLSDESC(S + A), SCHEMA(*P))
0xE203	R_AARCH64_AUTH_IRELATIVE	SIGN(Indirect(S + A), SCHEMA(*P))

17.1 Compatibility between relocatable object files

Relocatable objects in the same link-unit must agree on whether a GOT entry is signed. If there are AUTH and non-AUTH variant GOT generating relocations to the same symbol two GOT entries are required, one signed and one unsigned. While not a hard limitation many static linkers only support a single GOT entry per symbol. An implementation may choose to fault an AUTH and a non-AUTH GOT generating relocation to the same symbol, this would require all the GOT-generating relocations to a symbol to be signed or unsigned.

18 Appendix thoughts on encoding a signing schema

This section describes some of the trade-offs behind choosing a signing schema. It is not part of the ABI.

To create a signed pointer the run-time system needs to know the signing schema to use for the pointer. The object producer and static linker will need to communicate this via metadata; including at least:

- The Key, one of `IA`, `IB`, `DA`, `DB`. The `GA` key for signing of generic data is not exposed in this ABI.
- The constant discriminator value.
- Whether to combine address diversity with the discriminator.

In ELF we have the following places where we can encode this information via a combination of.

- The relocation code.
  - The relocation code could be used to communicate key and address diversity. There are not enough spare codes to describe a discriminator.
- The relocation addend.
  - AArch64 uses the `RELA` format which gives a 64-bit addend field. At a cost of limiting the size of the program, a number of bits of the addend could be reserved for communicating metadata.
- Writing data into the contents of the place being relocated.
  - The place is the operation `P` in relocation descriptions. It is derived from the r_offset field of the relocation.
  - When using `RELA` relocations, the contents of the place are ignored. The metadata could be written into the contents of the place and combined with the relocation.
- Default rules such as altering the behavior of existing relocations.
  - If there is a default signing schema for the GOT, and every GOT entry is signed with that schema we may not need any per-relocation encoding of the schema.

Some observations:

- Using the relocation code to encode key and address diversity would require 8 relocations to save 3-bits of metadata. If the `GI` key was supported by the ABI, 16 relocations would be needed to save 4-bits of metadata.
- Although ABI compliant ELF relocatable objects use `RELA` relocations, the type used in the link-unit is platform ABI. There are at least two documented relocation compression mechanisms (Android and `SHT_RELR`) and at least one platform that can support REL dynamic relocations.
  - In `SHT_RELR` the addend is written to the contents of the place like `SHT_REL` relocations.
- If the GOT is signed and a non default signing schema is used then the contents of the place of the relocation cannot be used to store the metadata as the linker creates the GOT entry.
- When not dynamic linking a static linker may choose to encode the pointer signing information in a custom encoding understood by the start-up code used.

19 Appendix Alternative ELF Marking Using GNU Program Properties

If a platform supports section `.note.gnu.property`, this can be used as the base for an alternative schema. The format of this section is defined in LINUX_ABI.

The following processor-specific program property types are defined:

Name	Value
GNU_PROPERTY_AARCH64_FEATURE_PAUTH	0xc0000001

The format of the data in `pr_data` is at least two 64-bit words, the first being a platform identifier, and the second being a version number specific to the platform identified in the first word. Consequently, the `pr_datasz` field must be at least 16. When `pr_datasz` is larger than 16, the remainder of the contents of `pr_data` are specific to the (platform id, version number). If `pr_datasz` is not a multiple of 8, `pr_padding` must be added so that PR_DATASZ + PR_PADDING is a multiple of 8 and the property remains 8-byte aligned.

The rules for the fields are the same as for the default marking schema: this ABI does not define the exact format of the platform and version identifiers, but reserves the combination of (platform, version) equal to (0,0) to represent an ELF file incompatible with this ABI.