关于 P6 系列处理器的微代码更新的研究

第一稿

作者：Jesus Molina, William Arbaugh
Park 大学， 2000 年 11 月
译者：albcamus <albcamus@whitecell.org>

摘要

P6 系列处理器（译注：P6 是 IA32 体系结构下的一种 Microarchitecture，属于 P6 系列的处理器实现有 Pentium Pro、Pentium II、Pentium III，以及 Pentium III Celeron/Xeon 变体）可以通过下载 Intel 提供的数据包来修正处理器的 BUG。 这一特性非常隐晦，而且没有文档化。 本文档试图解释微代码更新是怎样操作的， 可以修复什么 BUG， 如何被签名和加密的。

1。 介绍

微代码更新特性在参考文献[3]的第 8 章第 10 节有介绍。 微代码更新由 2048 个字节的数据组成， 其中 48 字节是头部数据， 剩下的 2000 字节是微代码本身。 为了避免歧义，本文将把这 2048 字节叫作「微代码更新」或「微代码」， 而把开始的 48 字节称为「头部」， 把剩下的 2000 字节称为「更新数据」。 「更新数据」是为了修复对处理器规范的背离（Intel 管这种背离叫「勘误表」(errata)。我们将用术语 BUG 或 errata 来指代这种背离）。

关于头部和更新数据的详细介绍， 将在本文档第 3 节出现。

微代码更新是通过 Intel 提供的软件工具完成的， 并且在重新引导时消失。 也就是说，你必须在每次机器起动的时候都更新微代码。

尽管 Intel 描述了微代码更新这一特征， 这种描述却是模糊的， 尤其在涉及到微代码所勘正的那些错误时。 另外， 微代码加了签名（我们将在第 7 部分详细讲述，参考资料[3]的 8.10 部分也有讲述）， 但在资料[3]中并没说这种加密/签名究竟是怎么执行的。Alexander Wolfe 在他在 eetimes[12]的文章， 以及后来在 byte.com[11]说：「根据所了解到的，我们得出结论： 数据块直接映射（更确切的说法是：解密后映射）到微代码本身。」 也就是说， 解密之后的数据块包含专门的微指令(microinstructions)。 微指令是处理器内部最小的操作原语， 控制着诸如门电路(gates)开关等专门动作， 和组成一条指令的微操作的序列。 本文档不打算讨论微指令。 但显然微指令是处理器为用户所见的最低层次。 而且，既然微代码能修正处理器的 BUG， 理论上它就可能给处理器引入新的 BUG。 由于包含在微代码中的微指令很晦涩难懂， 更新时可用空间又太小， 这里很难使用复杂的算法。

连同本《介绍》， 本文档分为 7 个部分。 在第 2 部分中， 我们将讲述 MSR 寄存器和 CPUID 指令， 这是理解微指令所需要的背景知识； 在部分 3, 我们描述微代码的基本结构； 在第 4 部分， 我们将象在部分 3 中那样来介绍驱动程序， 以及为测试需要而做的一些改动； 在第 5 部分描述一些由微代码勘正的 BUG； 第 6 部分讨论我们做的一些尝试，以及 Intel 可能通过哪些方式来加密微代码； 第 7 部分列举还有哪些工作可以改进， 并给出相关结论与文档。

最后， 本文档所描述的所有工作， 基于 Tigran Aviazan 提供的 microcode 驱动程序（在 linux 内核中）和 Simon Erzen 的工具[15]。 所用的内核版本是 Linux 2.4.0-test10.

2 背景

2.1 CPUID 指令

CPUID 指令是由 Intel Pentium 处理器引入的。 从那以后， 它成为 Pentium 以及更新的芯片标识自己的官方方法， 并被新一些的 Intel 486 和 AMD，UMC，Cyrix，NexGen 等处理器支持。

Intel 提供了一种很直接的方法来探测处理器是否支持 CPUI 指令， 该方法使用 EFLAGS 寄存器的第 21 比特， 如果它的值可以被软件改变， 那么 CPUID 指令就是可用的。

在执行 CPUID 指令之前， 你要先设置 EAX 寄存器的值， 这将为 CPUID 指令提供信息。 表 2.1 列出了 CPUID 根据不同的 EAX 值所执行的操作及其结果。

表 2.1 EAX 寄存器的指对 CPUID 指令的影响

参数	***CPUID 指令的输出***
EAX=0	EAX <- CPUID 识别的最大值
	EBX:EDX:ECX <- 制造商标识串
EAX=1	EAX <- 处理器签名
	EDX <- 特征标志
	EBX:ECX <- 为 intel 保留（不要使用）
EAX=2	EAX:EBX:ECX:EDX <- 处理器配置参数
3≤ EAX≤ 最大值值值值	Intel 保留
EAX>最大值	EAX:EBX:ECX:EDX <- 未定义数据（不要使用）

对 P6 家族的处理器来说， 制造商标识串是 GenuineIntel。

处理器签名这样编码：

EAX 寄存器

Intel 保留	类型	家族	模型	小步(stepping)
32 14	12	8	4	0

对 P6 系列处理器来说， CPUID 指令返回的值， 由表 2.2 给出了其编码方法。 表中的值均为十进制数， 第一个值是家族（P6 家族）， 第二个值是模型， 最后一个是小步（stepping）， 微代码的编码方式与 CPUID 返回结果的编码方式相同。

表 2-2 P6 家族处理器上 CPUID 指令的返回值：

6 - P6 家族

0 - Pentuim Pro 一个小步样本（译注：Step 和 Stepping 译为小步，是指在同一个核心的 CPU 上的不同产品，例如 Pentium III 可能有多个 steppings）

1 – 正规 Pentium Pro

1 – B0
2 – C0
6 – sA0
7 – sA1
9 – sB1

3 – Pentium II “Klamath” (063x),233 到 333 MHz, 覆盖了 P6 的 MMX(163x)

2 – tdB0(1632) – P II 覆盖了 PPro
3 – C0
4 – C1

4 – P55CT（覆盖 P54）

5 – Pentium II “Deschutes” (266 到 450 MHz)，Pentium II Xeon (400+ MHz) 和 Celeron w/o L2 缓存(266 到 300MHz) （不具有 L2 缓存的 Celeron 处理器，将有 CPUID 2 返回相关信息）

0 – dA0
1 – dA1
2 – dB0
3 – dB1

6 – Celeron “A” “Mendocino” (w/ 128KB 全速 L2 缓存) 或 Pentium II PE(256KB 缓存)

0 – mA0
5 -
a -

7 – Pentium III “Katmai”

2 – kb0
3 – kC0

8 – Pentium III “Coppermine”, Celeron w/SSE

1 – Ca2
3 – Cb0
6 – CC0

a – Pentium III Xeon “Cascades”

f – Pentium 4 (“Willamette”) 家族

除了在普通的寄存器中返回一些值之外， CPUID 指令还做了增强， 在 MSR 寄存器（Model Specific Register）中也返回一些值。 这种增强并没有改变 CPUID 指令的语义， 它只不过多做了一件事：把一个更新 ID 放在地址 08Bh 的 64 位 MSR 寄存器中。 如果没有提供更新， 则 MSR 寄存器的值不会被改变； 通常， 在执行 CPUID 指令之前先把 0 存入 MSR， 如果 CPUID 执行之后 MSR 的值还是 0, 则表示没有更新。

执行 RDMSR 指令之后， EDX 寄存器中存放了更新 ID， 它标识了修订的版本。 该值与 CPUID 返回在 EAX 中的值一起， 唯一标识了某个特定的微代码更新。 签名 ID 可以直接和 BIOS 更新的头部中的 Update Revision 字段比较， 来确认 BIOS 更新已被正确加载。 没有哪两个连续的 Pentium Pro 处理器小步会使用相同的签名 ID。 对更新来说， 不同的小步之间的区别由 CPUID 指令返回值来区分。 关于 CPUID 指令的更多信息请参考[2],[7],[6]和[14]。

2.2 MSR 寄存器

Pentium 和 P6 处理器都包含了模型相关寄存器（MSRs）， MSR 寄存器是高度未文档化的， 但既然 Intel 手册在附录 H 中提到了 MSR 寄存器， 那么关于 MSR 寄存器信息的反向工程就是可能的。 终于， Intel 在[3]中给出了关于 MSR 的文档， 在 8.2、15 和附录 A；当然这些文档还不够。 例如， 文档中说从地址 80000000h 到 FFFFFFFFh 的寄存器是不可用的， 但根据某个勘误（精确的说是勘误 E29） ，我们完全可以在非法地址上执行 WRMSR 和 RDMSR 指令。 Intel 似乎没有修正这个 BUG 的意图， 而只是想告诉用户不使用非法地址而已。 关于该 BUG 的更多信息可以在[9]中找到。

MSR 寄存器是实现相关的， 也就是说， 不保证 Intel 未来的处理器中它会具有同样的语义， 甚至不保证会提供它。 提供 MSR 寄存器是为了控制各种各样的硬件和软件， 包括性能监控计数器， 调试扩展以及其他杂项。 使用 RDMSR 和 WRMSR 指令可以读写 MSR 寄存器， 参考[2]。

如同在 2.1 部分所说的， 微代码更新使用了两个 MSR 寄存器来执行。

已文档化了的 MSR 寄存器列表在[10]中有描述。

3. 微代码结构

在《介绍》部分我们说过， 微代码是由头部和数据组成的。

表 3.1 中我们给出了微代码的结构。 头部划分成很多字段， 而更新数据被共享。 在表 3.2 中给出了头部中各个字段的简明解释。

表 3.1 微代码更新的结构

<table>
<tr><th colspan="4">字段</th><th>字节</th></tr>
<tr><td colspan="4">头部的版本</td><td>00H</td></tr>
<tr><td colspan="4">更新版本</td><td>04H</td></tr>
<tr><td colspan="4">日期</td><td rowspan="2">08H</td></tr>
<tr><td>月</td><td colspan="2">日</td><td>年</td></tr>
<tr><td colspan="4">处理器</td><td rowspan="2">0CH</td></tr>
<tr><td>Intel 保留</td><td>家族</td><td>模型</td><td>小步</td></tr>
<tr><td colspan="4">校验和</td><td>10H</td></tr>
<tr><td colspan="4">加载器版本</td><td>14H</td></tr>
<tr><td colspan="4">Intel 保留</td><td>18H</td></tr>
<tr><td colspan="4">数据</td><td>30H
7FCH</td></tr>
<tr><td>24</td><td>16</td><td>8</td><td>0</td><td>比特</td></tr>
</table>

表 3.2 各个字段的描述

字段名	偏移（以字节为单位）	长度（以字节为单位）	描述

头部版本	*0*	*4*	*微代码更新的头部版本*
更新修订号	4	4	更新的唯一的版本号。 处理器提供的更新签名指示了当前 CPU 内部的更新版本。 用来给 BIOS 认证更新并验证是否成功加载。该字段的值不能单独作为处理器小步的标识。
日期	8	4	该微代码更新的创建日期，二进制格式：mmddyyyy，月日年(例如，07/18/95 是 07181995h)
处理器	12	4	需要更新的处理器的家族，模型和小步。(例如 00000611h) 每个 BIOS 更新都是为某个家族、模型和小步的处理器特别设计的，BIOS 联合使用该字段与 CPUID 指令来探测处理器是否需要该次更新。 该字段的编码与 CPUID 指令的返回一一对应。
校验和	16	4	更新数据和头部的校验和。 用来验证头部和数据的完整性。 如果这 512 个双字的微代码的总和为零， 校验和就是正确的。
加载器版本	20	4	加载程序需要该字段才能正确加载更新。初始值为 0000001h。
Intel 保留	24	24	为将来的扩展保留
更新数据	48	2000	更新数据

Linux 的 microcode 驱动程序会检查头部中的字段。 由于它们只用来做标识， 且驱动程序检查之后就再也不管他们了（因此我们可以忽略）， 我们下面继续关注数据部分的结

构。

首先从 Intel 提供的 55 个微代码更新开始。 其中 5 个是为 Pentium Pro 的， 20 个为 Pentium II(Klamatch, Deschutes)， 8 个为 Pentium Celeron， 20 个为 Pentium III(Coppermine, Katmai)， 2 个为 Pentium III Xeon Cascades。 通过一字节一字节地相互比较这些微代码更新的数据部分的异同， 我们得到了结果。 前边我们说微代码的数据部分是不与其他更新共享的， 其实这偶而是有例外的。 这些例外是：平均每 256 字节中有 1 字节是共享的。 因此在一个微代码更新中， 大概有 8 个这样的地方。

——4 个 Pentium Pro 的微代码更新数据与其他的都不相同， 但是这 4 个之间同乡了最后的 1136 字节。 Pentium Pro 小步 B0（CPUID 指令的输出编码为 0611）则不与其它更新共享任何数据。

——42 个 Pentium Ii 和 Pentium Ii 更新数据共享了最后的 1056 字节。 它们与 Celeron 和 Pentium Pro 的微代码完全不同。

——8 个 Celeron 更新与其他的更新数据完全不同。

我们将在第 6 部分讨论这些结果。

4. 驱动程序的详细说明

Linux 的 microcode 驱动程序是这样工作的：

- 检查头部。 包括为处理器/小步选择正确的微代码， 拒绝掉不匹配的微代码， 拒绝掉不匹配处理器小步的微代码， 拒绝掉其版本比当前处理器中的还低的微代码， 拒绝掉校验和不正确的微代码（在第 2 部分我们已经看到了，校验和就是微代码的总和，应该为零）。
- 初始化一些寄存器， 以调用 WRMSR
  - EAX 包含更新数据的线性地址
  - EDX 为 0
  - ECX 为 79h
- 调用 WRMSR（已经更新了微代码数据）
- 初始化寄存器， 以调用 WRMSR
  - EAX 为 0
  - EDX 为 0
  - ECX 为 8bh
- 调用 WRMSR（已经将 8bh 处的 MSR 寄存器清零）
- 初始化寄存器， 以调用 CPUID
  - EAX 为 1
- CPUID
- 初始化寄存器， 以调用 RDMSR
  - ECX 为 8bh
- 调用 RDMSR

如果没有加载微代码， MSR 保持不变（我们在 CPUID 指令之前先把它清零了）。如果成功加载了， 寄存器的低 32 位就包含了标准的模型/小步信息， 高 32 位包含了微代码更新 ID。

Intel 提供的更新数据是文本的形式， 所以我们必须在加载之前， 首先把这些文件

「翻译」成二进制。

为了执行测试， 我们稍稍改写了驱动程序。 改写以后的驱动程序不检查头部， 只观察 8bh 地址的 MSR 在 CPUID 后是否改变， 来判断微代码是否成功加载。 这种方法不用担心当加载错误的微代码时， 其头部数据格式发生什么变化。

5.微代码能修正哪些 BUG

研究微代码的一个有趣而关键的问题是： 微代码究竟能修正什么 BUG？ 关于 Intel 所分发的每个微代码是针对哪些 BUG 的， 它从未提供任何说明。

Intel 为它们的每个处理器分发叫作<处理器规范更新>(Processor Specification Updates)的文档。 文档[4],[5],[6]包含了 Intel 对这些规范所作的更改， 列出了所有的 BUG（至少是 Intel 愿意文档化的那些）， 并有相关的描述， 问题是否可能包含在处理器中的暗示， 相关工作状态是什么（Fix 表示正打算修正；Fixed 表示已经修正了；NoFix 表示不打算修正该 BUG）。

在某个勘误中 Intel 说道， “BIOS 代码包含针对该 BUG 进行的相关工作， 这是可能的。” 乍一看似乎是表示 BUG 是由微代码修正的， 然而令人吃惊的是， 微代码的数量比 BUG 的数量还多！ 例如， 在 intel 于 2000 年 11 月发放的 Pentium III 处理器规范更新中，描述了错误 E34,E35,E44,E46,E54,E56,E58,E64 已由相关微代码更新修正； Coppermine 处理器的 Cb0 小步（CPUID 为 0683）， 列出了 E44,E46,E54,E58,E64， 有至少 5 个微代码； 然而 Coppermine 处理器的 Ca2 小步（CPUID 为 0681）， 列出了四个为微代码更新修正的 BUG（E44,E46,E58,E64）, 但我们至少能找到 5 个微代码。 另外， Coppermine 处理器的 CC0 小步（CPUID 为 0686）， 我们只看到一个错误：E46 ， 然而注意少有 3 个不同的微代码（几乎是同时分发，一个在 05/04/00, 另外两个在 05/05/00）， 为什么只修正 1 个 BUG，我们就需要 3 个不同的微代码？ 答案最可能是： 一些 BUG 是 Intel 不愿文档化的， 或者有些 BUG 未必是那么容易由微代码更新解决， 或者二者都有。 事实上， Pentium Ii 处理器的 E19 和 E20 是由同一个微代码修正的， 但只有 E20 是文档化的。

最后， 我们并不清楚 Intel 如何操纵 BIOS 的相关工作。 Pentium III 的 E34,E35,E54 标记为 Fixed， 但 E56,E44,E46,E58,E64 标记为 NoFix。 不清楚 Intel 是否假设了微代码更新是对 BUG 的合法修正(?什么意思?)。

由 BIOS 相关工作和微代码更新解决的完整 BUG 列表， 可以在本项目的 Web 页面找到， 参考第 7 部分。

6. 微代码的加密系统分析

分析微代码的加密系统， 其难度简直与试图理解由外语写的随机信息差不多。
然而， 我们可以利用一些暗示。

- Pentium Pro 的微代码和 Pentium Ii 的微代码， 以及 Pentium III 的微代码共享了一部分数据。 这可能是因为它们共享了对 BUG 的补丁， 也可能是它们使用了相同的微代码加载代码。 这样， 加载代码就不必内置到处理器中。 由于 Celeron 是个不同的设计， 便只能把加载代码内置到处理器中。
- 微代码内部必须包含关于模型/小步的信息， 和微代码版本的信息， 这由 CPUID 返回。 这种信息或者由微代码数据，或者由加密系统自己提供。 （例如，为每个处理器小步使用不同的 key）

- 处理器拒绝伪造的微代码
- 处理器拒绝其模型/小步与自己不同的微代码
- 新的微代码可能包含有这样的补丁： 它修正旧的微代码修正的 BUG
- 加密系统不可能是很复杂的算法， 这样才不会浪费核心(die)空间。 当然这只是猜测。

签名微代码的两种最可能方式是：

- 加密全部或部分的微代码。 加密后的数据包含一个 CRC 值， 在加密数据后， 处理器检查它的值。
- 使用签名， 例如 CRC 或 MAC

假设 Intel 加密了整个微代码， 可能的选择为：

- 使用同样的 key 加密所有的微代码
- 根据处理器小步加密每一个微代码， 为每个处理器小步使用不同的 key。

如果 Intel 是使用相同的 key 加密所有的微代码， 我们比较不同的微代码之间的共享数据时， 就应该找到更多的相同部分。 唯一可能避免共享数据的方式是， 不考虑微代码指令的任何次序。 两个数据块之间的卷积(convolution)将使得我们看到更多信息， 反映了相关的模式。 并且， 如果他们并没有加密数据部分， 而只是使用 MAC 或 CRC， 在微代码间就肯定得有一些匹配， 而且对频率信息做分析将得到有趣的结果。

如果 Intel 为每个模型/小步使用了不同的 key， 则相同的模型/小步的微代码之间就应该出现一些重复的模式。 我们目前没有在微代码数据发现任何像是针对相同模型/小步的重复模式。 因此， 可能使用了一种混淆机制， 例如使用微代码的版本作为混淆。 这种方案下， 所有微代码共享的部分的最后一部分必定是加载代码。

7. 将来的工作和结论

已经完成的工作有：

- 对 tigran Aviazan 和 Simon Erzen 的驱动程序做修改， 以不检查头部
- 把微代码更新转化为二进制文件， 依处理器/小步排序
- 在数据上执行测试
- 在数据上测试被拒绝的情形
- 搜索 Pentium 规范更新文档， 查找那些可以用 BIOS 相关工作解决的 BUG。

将来的工作有：

- 数据的出现频率分析
- 在不同的微代码间寻找重复出现的模式， 它们并不依次序
- 寻找微代码和 BUG 的关系
- 观测使用 MSR 寄存器加载微代码后， CPU 的行为

所有的工具和进一步工作可以在 Web 页面上找到:
http://www.jesusmolina.com/projects/microcode
或者镜像站：
http://wam.umd.edu/~chus/microcode

8. 参考文献

1. IA-32 Intel® Architecture Software Developer's Manual, Volume 1: Basic Architecture. Intel Corp
2. IA-32 Intel® Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual. Intel Corp
3. IA-32 Intel® Architecture Software Developer's Manual, Volume 3: System Programming Guide. Intel Corp
4. Pentium® III Processor Specification Update. Intel Corp
5. Pentium® II Processor Specification Update. Intel Corp
6. Pentium® Pro Processor Specification Update. Intel Corp
7. Intel Processor Identification and the CPUID Instruction. Intel Corp, April 1998, AP-485:Application note
8. Intel® Pentium® 4 Processor Identification an the CPUID Instruction. Intel Corp Revision 001, July 2000
9. "Pentium Model\u2212Specific Registers and What They Reveal". Ralf Brown, Revision 1.0, October 11, 1995 http://x86.org/articles/p5msr/pentiummsrs.htm
10. "Model\u2212Specific Registers",Release 60. Ralf Brown, January1999 http://chip.ms.mff.cuni.cz/~pcguts/cpu/msr.txt
11. "Intel Sneaks In Software\u2212Upgrade Feature". Alexander Wolfe, October 2000 http://www.byte.com/column/BYT20001016S0006
12. "Intel preps plan to bust bugs in Pentium MPUs". Alexander Wolfe, EETimes, Issue 960, January 1997
13. "The Pentium collects lots of information about code execution, and now you can get access to it". Terje Mathisen http://www.byte.com/art/9407/sec12/art3.htm
14. "Identification of x86 CPUs with CPUID support", Grzegorz Mazur http://grafi.ii.pw.edu.pl/gbm/x86/cpuid.html
15. "Intel P6 Microcode Update Utility for Linux", Simon Erzen, Tigran Aviazan http://www.urbanmyth.org/microcode/

WSS(Whitecell Security Systems)，一个非营利性民间技术组织，致力于各种系统安全技术的研究。坚持传统的 hacker 精神，追求技术的精纯。

WSS 主页：http://www.whitecell.org/

WSS 论坛：http://www.whitecell.org/forums/