# iDEV: Exploring and Exploiting Semantic Deviations in ARM Instruction Processing

Shisong Qin
Tsinghua University
Beijing, China
qss19@mails.tsinghua.edu.cn

Chao Zhang*
BNRist, Tsinghua University, and
Tsinghua University & QI-ANXIN Group JCNS
Beijing, China
chaoz@tsinghua.edu.cn

Kaixiang Chen
Tsinghua University
Beijing, China
ckx18@mails.tsinghua.edu.cn

Zheming Li
Tsinghua University
Beijing, China
lizm20@mails.tsinghua.edu.cn

## ABSTRACT

ARM has become the most competitive processor architecture. Many platforms or tools are developed to execute or analyze ARM instructions, including various commercial CPUs, emulators, and binary analysis tools. However, they have deviations when processing the same ARM instructions, and little attention has been paid to systematically analyze such semantic deviations, not to mention the security implications of such deviations.

In this paper, we conduct an empirical study on the ARM *Instruction Semantic Deviation* (ISDev) issue. First, we classify this issue into several categories and analyze the security implications behind them. Then, we further demonstrate several novel attacks which utilize the ISDev issue, including *stealthy targeted attacks* and *targeted defense evasion*. Such attacks could exploit the semantic deviations to generate malware that is specific to certain platforms or able to detect and bypass certain detection solutions.

We have developed a framework iDEV to systematically explore the ISDev issue in existing ARM instructions processing tools and platforms via differential testing. We have evaluated iDEV on four hardware devices, the QEMU emulator, and five disassemblers which could process the ARMv7-A instruction set. The evaluation results show that, over six million instructions could cause dynamic executors (i.e., CPUs and QEMU) to present different runtime behaviors, and over eight million instructions could cause static disassemblers yielding different decoding results, and over one million instructions cause inconsistency between dynamic executors and static disassemblers. After analyzing the root causes of each type of deviation, we point out they are mostly due to ARM unpredictable instructions and program defects.

*corresponding author


*ISSTA '21, July 11–17, 2021, Virtual, Denmark*

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464842

## CCS CONCEPTS

• **Security and privacy → Software and application security**.

## KEYWORDS

ARM Instruction Processing; Semantic Deviation

**ACM Reference Format:**
Shisong Qin, Chao Zhang, Kaixiang Chen, and Zheming Li. 2021. iDEV: Exploring and Exploiting Semantic Deviations in ARM Instruction Processing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464842

## 1 INTRODUCTION

ARM has been widely used in all kinds of electronic devices, e.g., tablets, mobile phones, and many IoT devices. In addition to the low price and power consumption, the essence for ARM's success is its unique business model - any manufacturer with a license could implement its own ARM design in silicon. Such a model brings the prosperity of ARM-powered devices but also brings minor deviations in different products supporting ARM instructions, since different manufacturers may use different solutions to implement features defined in the ARM ISA specification. Not to mention that the ARM ISA specification also has ambiguity. Specifically, the specification leaves room for *unpredictable* instructions whose behavior is uncertain, thus undocumented instructions could be introduced by manufacturers.

In addition to hardware implementation deviations, software that processes ARM instructions could also have deviations. Disassembling is the most direct way to process instructions. Based on the disassembly results, applications could further perform (1) static binary analysis, e.g., static binary instrumentation [1], binary lifting [2, 36], reverse engineering [35, 40, 42] and binary similarity detection [18]; and (2) dynamic analysis, e.g., emulation [3] and dynamic binary instrumentation (DBI [3, 4, 17]). Developers of these disassemblers or binary analysis tools could misunderstand the specification or fail to keep pace with the latest specification, and thus introduce deviations too.

In general, such deviations could leak information related to execution platforms or analysis tools, enabling adversaries to distinguish execution environments and launch targeted attacks, e.g.,

Stuxnet [18]. However, little attention has been paid to systematically analyze such ARM instruction semantic deviations nor their security implications and root causes. Therefore, in this paper, we conduct an empirical study on the semantic deviation of ARM instruction for the first time.

Firstly, we systematically summarize the semantic deviation of binary instruction, and present the *instruction semantic deviation* (ISDev) issue. We categorize the ARM ISDev issue into several categories, including deviations between execution behaviors, between disassembly results, and between their combinations. Then we analyze the security implications behind these deviations and present several attack scenarios, where adversaries could generate malware specific to certain platforms, or able to detect and bypass certain detection solutions, with corresponding proof-of-concept attacks.

Secondly, we propose the iDEV framework to systematically explore the ISDev issue in processing ARM instructions. It utilizes differential testing to evaluate ARMv7-A instruction set supported by different platforms and tools, and identify ISDev cases when these targets yield different results. Our iDEV framework explores ARM instructions in the ARMv7-A instruction set and aggregates instructions that have similar semantics to reduce the size of the instruction set to test. By adopting two refinement policies, iDEV could explore almost all types of ARM instructions by only testing 1/128 of the original instruction set space. For each instruction, iDEV collects signals triggered by different platforms and records the normalized decoding results of different disassemblers. iDEV then utilizes differential testing techniques [33] to extract deviations between these platforms and tools. Finally, iDEV helps us further explore the root cause of ARM ISDev cases.

Thirdly, we have implemented a prototype of iDEV, and have evaluated it on four hardware devices, the QEMU emulator, and five disassemblers. The results show that, over six million instructions (more than 18% of all instructions) could cause dynamic executors (including hardware devices and the QEMU emulator) to present different runtime behaviors, and over eight million instructions (more than 26% of all instructions) could cause static disassemblers yielding different analysis results, and over one million instructions (more than 3% of all instructions) cause inconsistency between disassemblers and executors. By analyzing such ISDev cases, we point out the main root cause of such deviations are ARM unpredictable instructions and program defects.

In summary, we make the following contributions in this paper:

- We summarize and categorize the instruction semantic deviation problem, and analyze the security implications of ISDev issue in devices and tools supporting ARMv7-A ISA. We propose several attack scenarios where adversaries could generate malware specific to certain execution platforms or able to detect and bypass certain detection solutions.
- We present iDEV, a framework to systematically explore ISDev cases in ARMv7-A instruction processing devices and tools, including hardware, emulators, and disassemblers.
- We have evaluated iDEV framework on four hardware devices, the QEMU emulator, and five disassemblers, then have measured the prevalence of ARM ISDev problem in real-world. The results show that a huge number of ARM instructions could result in semantic deviations, and we further point out the root causes behind these deviations.

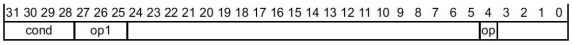



**Figure 1: General ARM instruction encoding.**

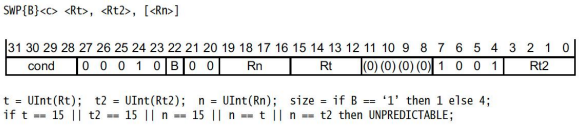



**Figure 2: Encoding and pseudocode description of the *SWP, SWPB* instruction shows they have unpredictable variants.**

## 2 BACKGROUND

### 2.1 ARMv7-A Instruction Set Encoding

ARM has become the most widely used instruction set architecture (ISA). It is worth noting that although architecture (e.g., ARMv3 to ARMv8-AArch32) share common instruction sets and programmer's models with backward compatibility, they may differ from each other in execution states and supported features. In this paper, we choose the ARMv7 version as our objective of the research.

ARMv7-A is the *Application profile* of ARMv7 architecture, aiming at the market with higher performance needs. It belongs to the family of reduced instruction set computing (RISC) architectures.

A typical ARM instruction encoding is shown in Figure 1. In the instruction pipeline, each bit of the encoding may affect the behavior of the processor. But some of the bits are strictly regulated under ARM's specification and are used to control the operations and conditional executions. Specifically, *bit 4* and *bits[25:31]* mainly determines the semantic of an ARM instruction, and *bits[31:28]* specifies the condition attributes that determines if the processor will execute it. Prior to execution, the processor compares the condition attributes with the condition flags in the *CPSR* register, and the instruction will be executed only if the condition is matched.

### 2.2 ARM Unpredictable Instruction

In ARM ISA specification, some instructions are defined as *UNPREDICTABLE*. First, in the encoding diagram of a certain kind of instruction, some bits are suggested to be set to fixed values - either (0) or (1). If these bits are not set as the specification denotes, the behavior of this instruction is UNPREDICTABLE. Second, some instructions are explicitly labeled as UNPREDICTABLE in their pseudocode description, if certain conditions are met. Figure 2 shows the encoding diagram of *SWP,SWPB* instruction, which contains UNPREDICTABLE conditions.

Although the ARM manual claims unpredictable instructions can be implemented as UNDEFINED, it implies that the real implementation of unpredictable instructions could vary from one manufacturer to another. We can see that, whenever an unpredictable instruction is executed, it will trigger an unreliable action that does not attain consensus from other implementations.

Therefore, we believe deviations, brought by different implementations of unpredictable instructions, are prevalent. These deviations will also confuse software that analyzes ARM instructions.

## 2.3 Instruction Semantic Deviation Problem

Previous work [19, 24, 25, 37, 38, 41] mainly use differential testing and analysis to explore the instruction semantic deviation problem. Differential testing is a popular testing technique that attempts to detect potential issues in system implementation, by providing the same input to a series of similar applications, and observing differences in their executions. Any discrepancy among the program behaviors on the same input is marked as a potential bug.

In 2010, Paleari et al. [38], firstly applied differential testing to multiple x86 disassemblers and revealed numerous bugs in these disassemblers. Nathan et al. [24] performed differential testing on instruction decoders for multiple architectures in 2017. Recently, the Trail of Bits team released their differential fuzzing framework mishegos [37] towards x86_64 architecture decoders. Other than disassemblers, differential testing is also capable of other kinds of software testing, i.e., Soomin et al. [25], excavated semantic issues of Intermediate Representations with elaborative differential testing. In addition to traditional differential testing for software implementation, Domas et al. [19] compared the execution and disassembly results of x86 ISA to dig out hidden instructions from the Intel manual, PROTEUS [41] observed ARM instruction execution deviation between the emulator and physical CPUs. These studies are also based on the idea of differential analysis.

In this paper, we still utilize differential testing to explore semantic deviations in processing ARM instructions, not only within disassemblers but also within execution devices and between them.

# 3 STUDY ON ARM ISDEV PROBLEM

## 3.1 Definition and Classification of ISDev

Informally, we use `ISDev` to represent all instructions that could cause semantic deviation. Specifically, `ISDev` cases could be further classified into the following 3 types.

- Type1: *Execution Deviation.* For instructions that would cause different run-time behaviors on various execution platforms including physical and emulated CPU, are classified as type1, namely execution deviation.
- Type2: *Disassembly Deviation.* For instructions that would yield different decoding results on various disassemblers are classified as type2, namely disassembly deviation.
- Type3: *Execution-Disassembly Deviation.* For instructions that have inconsistent results between execution platforms and disassemblers, we divide them into two subcategories: Type3-1 deviation (executable but not decodable) and Type3-2 deviation (decodable but not executable).

## 3.2 Security Implication of ARM ISDev

In this part, we analyze the security implications of the aforementioned `ISDev` problem, and demonstrated several proof-of-concept attacks that exploit ARM `ISDev` based on some specific cases.

*3.2.1 Type1: Execution Deviation Exploits.* Type1 deviation in instruction processing enable adversaries to recognize the instruction execution platform. Since different platforms yield different runtime behaviors of these instructions, adversaries could monitor a group of instructions' executions to infer which platform these instructions are running on. By exploiting the leaked platform information, adversaries could launch a series of targeted attacks.

**Listing 1: A PoC malware able to detect and bypass the virtual environment QEMU.**

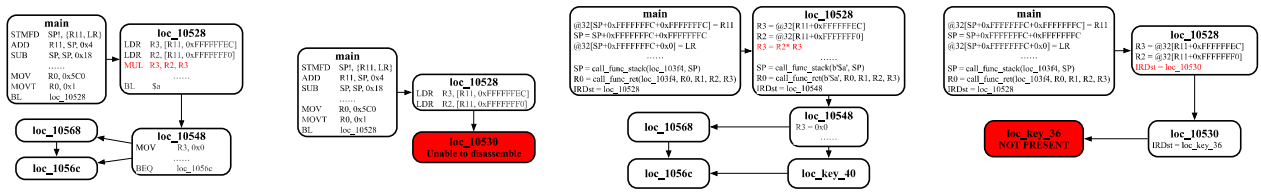
```
void check_environment() {
    register_handler(qemu_handler);
    // 0xe68000b0 is an unpredictable instruction
    // executable on Pi 2B, but triggers SIGILL on QEMU
    __asm__ __volatile__(".word 0xe68000b0\n");
    return;
}
void qemu_handler(int sig_num, siginfo_t* siginfo, void* myact
    ) {
    // skip execution of malicious code on QEMU
    ((ucontext_t*)myact)->uc_mcontext.arm_pc +=
                SIZE_OF_MALICIOUS_CODE;
}
void register_handler(void (*handler)(int, siginfo_t*, void*))
    {
    struct sigaction act;
    act.sa_sigaction = handler;
    sigaction(SIGILL, &act, NULL);
}
```

First, adversaries may utilize it to detect virtual execution environment to bypass malware analysis or application emulation, i.e., launching *targeted defense evasion*. Specifically, they could exploit Type1 deviation to infer virtual environments like QEMU, and stop executing malware code in them but keep executing in hardware environments. Note that, PROTEUS [41] has explored the possibility of recognizing Android emulators based on instruction profiles. But it not treated deviations in unpredictable instruction execution as a reliable exploit. However, we thought the semantic deviations in a large number of unpredictable instructions are still useful.

Moreover, adversaries could also utilize it to detect physical devices to launch *stealthy targeted attacks*. Specifically, they could test runtime behaviors of a group of instructions that have Type1 deviation, for fingerprinting the target execution platform. Then, they only continue executing malware code in specific hardware but stopping executing in other devices. In this way, these malware samples are much harder to discover. Targeted attacks like Stuxnet [7] could become more stealthy.

**Proof-of-Concept Attacks.** Listing 1 shows a proof-of-concept malware, which exploits Type1 deviation to recognize virtual execution environment (i.e., QEMU) and skip executing malicious code if detected. Specifically, an attacker could embed an instruction `0xe68000b0`, which executes on hardware devices as *SEL* but raises a SIGILL signal in QEMU, into the malware sample. This instruction will raise a signal and triggers the predefined handler which skips the malicious code in QEMU. As a result, this malware could bypass dynamic analysis or detection solutions based on QEMU. Similarly, adversaries could exploit execution deviations to fingerprint physical execution platforms, and execute malware only on specific devices, similar to what Stuxnet did.

*3.2.2 Type2: Disassembly Deviation Exploits.* There have been some previous work [24, 37, 38] that used differential testing to discover errors in disassemblers. Therefore, the exploits of disassembly deviation are actually the exploits of such errors. Disassembler errors can be classified into the following three categories: (1) *Under Decoding*. It represents errors that disassemblers fail to recognize or decode valid instructions. (2) *Incorrect decoding*. It represents errors that disassemblers incorrectly decode valid instructions. (3) *Over decoding*. A disassembler may *successfully* decode instructions which are explicitly marked as UNDEFINED in ARM manual. Based on the type of disassembler errors, there are three kinds of exploits.




(a) Disassembly of original program (b) Disassembly of modified program (c) IR of original program (d) IR of modified program

**Figure 3: Miasm's disassembly result and IR result after binary lifting.**

**Listing 2: An Incorrect decoding PoC which could deceive analysis solutions (against Miasm).**

```
// Source code:
void main() {
    int a = 5;
    //  0xe6012340 = encoding of "STR R2, [R1], -R0 ASR 0x6"
    __asm__ __volatile__(".word 0xe6012340\n");
}
; Disassembly result from Miasm
MOV R3, 0x5
STR R3, [R11, 0xFFFFFFF8]
STR R2, [R1], R0 ASR 0x6 ; Incorrect Decoding
MOV R3, 0x0
```

- *Under Decoding:* In this case, a disassembler fails to recognize or decode a valid instruction. It in general will *truncate the disassembly result or halt or hang the disassembly process*, yielding an incorrect or incomplete disassembly result. Adversaries could craft malware samples to break analysis tools that are based on this disassembler.
- *Incorrect Decoding:* Instructions can be misinterpreted by the disassembler. Therefore, it yields wrong disassembly results and misleads analysis tools based on the disassembler. Adversaries could thus craft malware samples *to deceive analysis solutions*, e.g., symbolic execution and taint analysis, and bypass analysis or detection solutions based on the disassembler.
- *Over Decoding:* Disassemblers may wrongly decode invalid instructions. Therefore, it will continue decoding after the over-decoding instruction, and get a control flow graph (CFG) different from the actual one. As a result, it faces the *same consequences as the aforementioned incorrect decoding case.* Moreover, adversaries could utilize exception handling mechanisms, e.g., registering a handler to process exceptions caused by the over-decoding instruction, to *hide the real control flow graph from static analysis* and obfuscate the binary.

**Proof-of-Concept Attacks.** Listing 2 shows a PoC attack utilizing disassembler errors. Miasm [42] incorrectly decodes operands of the *STR* instruction. Specifically, Miasm overlooks one flag bit in the instruction, which is used to specify whether the operand is positive or not, leading to a wrong disassembly result. Hence, such deviation would deceive analysis solutions based on Miasm.

*3.2.3 Type3: Execution-Disassembly Deviation Exploits.* As aforementioned, there are two categories of Type3 ARM `ISDev` cases, i.e., Type3-1 deviation (executable but not decodable) and Type3-2 deviation (decodable but not executable). Different categories have different security implications.

**Type3-1 Security Implications.** For executable but not decodable instructions, adversaries could exploit them to launch malicious actions on target platforms but confuse binary analysis tools, which is also a type of *targeted defense evasion* attack. Besides, these

**Listing 3: A PoC malware exploiting Type3-1 semantic deviations (executable but not decodable).**

```
int main() {
    int a, b, c;
    scanf("%d %d", &b, &c);
    a = b * c;
    return malicious_code();
}
; Disassembly result from Miasm;
...
LDR R2, b_addr
LDR R3, c_addr
; executable and decodable, generated by gcc
-MUL R3, R2, R3 ; word: 0xe0030392
; not decodable by Miasm, modified by adversaries
+MUL R3, R2, R3 ; word: 0xe0031392
STR R3, a_addr
```

instructions can also be exploited by adversaries to create variants of malware by simply rewriting a few bits in malware samples.

**Proof-of-Concept Attacks.** As shown in Listing 3, a PoC malware sample could utilize an executable but undecodable instruction to evade analysis tools based on the binary analysis framework Miasm [42]. Specifically, the statement `a = b * c` in the program would be assembled to a *MUL* instruction by compilers. In this case, it would be `MUL R2, R3`, encoded as `0xe0030392`. According to the encoding diagram of the *MUL* instruction, if any bit in its bits[15:12] is set to 1, the instruction becomes unpredictable.

However, these unpredictable instructions derived from the *MUL* instruction cannot be decoded by the binary analysis framework Miasm, and these unpredictable instructions can still be executed normally on all platforms we selected for evaluation. Besides, after further monitoring the change of the register values before and after the execution of this instruction, we found that the unpredictable instruction has the same semantics as the original *MUL* instruction. Therefore, an attacker could rewrite the encoding of *MUL* instruction from `0xe0030392` to `0xe003[1-f]392` to truncate the disassembly result.

As shown in Figure 3(b), Miasm fails to disassemble the target function. Further, its binary lifting engine also fails to lift the binary to its intermediate representation (IR), as shown in Figure 3(d). Meanwhile, the modified instruction can still be executed on devices normally. In this way, the malware could evade static analysis of Miasm while keeping its functionality.

It is worth noting that, adversaries in general only need to rewrite several bits of target instructions to launch such attacks, making such attacks simple but effective.

**Type3-2 Security Implications.** For the second class of Type3 deviation, those instructions could be decodable but not executable. Adversaries could exploit them to deceive binary analysis tools with wrong disassembly results, but execute malicious code on target

**Listing 4: A PoC malware exploiting type3-2 semantic deviations (decodable but not executable).**

```
int main() {
    register_handler(malicious_code);
    // 0xe1800090 is an unp inst which triggers SIGILL
    // on raspi2 and IDA decode it normally
    __asm__ __volatile__(".word 0xe1800090\n");
    fake_cfg();
}
void  malicious_code() {
    true_cfg();
}
void register_handler(void (*handler)(int, siginfo_t*, void*))
{
    struct sigaction act;
    act.sa_sigaction = handler;
    sigaction(SIGILL, &act, NULL);
}
```

executors by hooking execution exceptions. In other words, they could also launch *targeted defense evasion* as previous attacks.

**Proof-of-Concept Attacks.** Listing 4 shows a PoC that exploits an instruction decodable by IDA Pro but not executable on raspberry Pi 2B. It can conceal malicious code in signal handlers, and presents a different control flow graph to analyzers based on IDA Pro.

In this PoC, the attacker firstly registers a handler for the SIGILL signal and places his malicious code in the handler. After completing the signal registration, the attacker can put an inline assembly statement anywhere in the program to execute the `0xe1800090` instruction. This instruction is an unpredictable instruction that is not executable on Pi 2B but can be decoded by IDA Pro [40] as `orr r0, r0, r0 lrl r0`. Therefore, IDA Pro continues to decode and generate a wrong control flow graph. However, when running in real executors, a SIGILL signal will be triggered by this instruction, and then malicious code in the registered handler will be executed. Thus, the use of such decodable but not executable instruction could help attackers conceal malicious code in CFG.

This PoC is similar to the virtual environment detection demo shown in Listing 1, but is a completely different attack scenario.

# 4 APPROACH

Based on our study about ISDev problem, we develop a framework iDEV to explore different types of ARM ISDev cases in ARMv7-A ISA, while analyzing the root cause of them.

## 4.1 Overview Design

Figure 4 depicts the workflow of iDEV. At the high level, iDEV firstly preprocesses the instruction set to aggregate instructions with similar semantics, thereby reducing the instruction set space that needs to be explored. Then differential testing is utilized to find out all kinds of ISDev cases. In the end, we look into these deviations and perform further root cause analysis. In general, iDEV takes the refined ARM instruction set as input and extracts ARM ISDev cases, then outputs analysis reports with the root cause of such deviations.

**Preprocessing.** Although the number of ARM instruction is much smaller compared with the variable-length instruction sets such as x86, its entire instruction set space is still huge (e.g., the size of 32-bits fixed ARM instruction set space is $2^{32}$). It thus brings huge challenges to systematically explore all ARM instructions. Therefore, we propose a solution to refine the ARM instruction set space. Eventually, iDEV aggregates instructions with similar semantics to reduce the size of instructions set as 1/128 of the original instruction set, following the encoding rules documented in ARM ISA manual [15]. The details are discussed in Section 4.2.

**Differential Testing.** iDEV utilizes differential testing to find ISDev cases in execution platforms and disassemblers. First, iDEV feeds the refined ARM instruction set to ARM 32-bit architecture execution platforms to perform dynamic execution testing. iDEV records the signal triggered by each instruction when it is executed and compares the results among different platforms. It then reports Type1 deviation cases for further root cause analysis. The approach details of dynamic execution testing are described in Section 4.3. Then, iDEV feeds the refined instruction set into multiple disassemblers and then compares the unified output from them to perform differential analysis. Similarly, iDEV retains the Type2 deviation cases to apply a further analysis. The details are shown in section 4.4. In addition, iDEV further cross-checks the results between dynamic testing platforms and static analysis tools, to detect Type3 deviation cases, then perform root cause analysis as well.

**Root Cause Analysis.** After differential testing, the ARM ISDev cases will be analyzed manually with the ARM specification to extract the root causes of these deviations. Then finally iDEV would output analysis reports about these cases.

## 4.2 Instruction Set Preprocessing

To thoroughly explore ISDev problem of ARM instruction processing tools, we need to analyze as many instructions as possible.

However, the search space of ARM 32-bit fixed-length instructions is $2^{32}$, more than four billion instructions. Although it is much smaller than the search space of CISC architectures, e.g., x86 has over $10^{36}$ instructions, it is still impractical to test all ARM instructions within reasonable time, not to mention mobile devices of ARM architecture tend to have small memory and weak computing capability. To make the further analysis practical, iDEV first needs to refine the instruction set before starting the testing phase.

Based on the encoding pattern detailed in ARM manual [15], an ARM instruction has three components, the condition field, opcode field, and operand fields. The opcode field represents the overall semantic of an instruction, hence it should be explored as fully as possible. Thus, iDEV tries to refine the condition field and operand fields to reduce the size of the instruction set space to explore.

**Condition Field Refinement.** As shown in Figure 1, a large number of ARM instructions support the feature of conditional execution, i.e., they will only be executed when the specified condition is met. These instructions use bits[31:28] as *Condition Code* to represent their execution condition. Therefore, iDEV first simply refines the instruction set by fixating the *Condition Code*, and only tests instructions whose *Condition Code* is set to *0xe*, which means they would be executed unconditionally. In this way, it can not only reduce the size of instructions set, but also ensures the chosen instruction will be executed (instead of skipping) during testing.

**Operand Field Refinement.** Regardless of condition code, instructions with the same opcode could have different operands. To further reduce instructions set, iDEV tries to aggregate instructions based on opcodes. However, there is no clear boundary between opcode and operand fields in the instruction encoding. The ARM ISA manual [15] states that the opcode of ARM instructions is

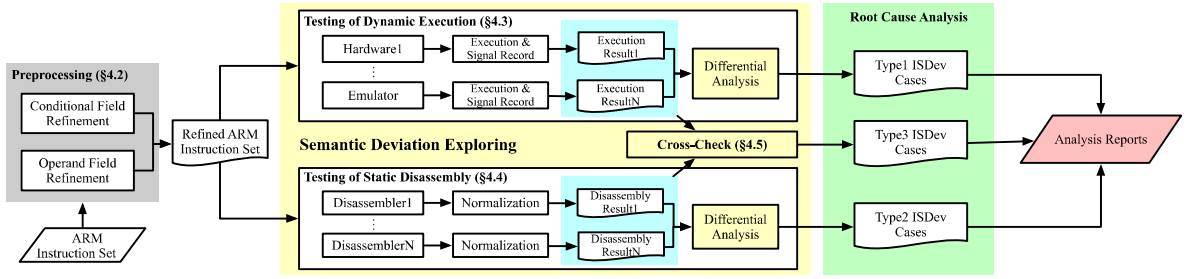



**Figure 4: Workflow of iDEV. It first preprocesses the ARM instruction set to reduce the size to explore, then performs differential testing to collect ISDev cases and analyzes root causes of these deviations, and finally outputs analysis reports.**

mainly determined by bits[31:25] and bit 4. However, based on our observations, the semantic of an ARM instruction is affected by several more bits. To locate a specific instruction encoding, we actually need to go through a multi-level table lookup, and several bit positions are used to determine its semantics.

Fortunately, we found that the lowest four bits of almost all ARM instructions are used as operands which do not affect instruction semantics. Therefore, we selected the extreme values *0x0* and *0xf* to fixate these 4 bits, thereby further reduce the instruction set.

**Preprocessing Results.** After these two refinement policies, the entire ARM instruction set is reduced to 1/128 of the original, while covering almost all ARM instruction semantics. Here we have only $2^{25}$ (about 33 million) instructions left to test, enabling low performance devices to perform dynamic testing.

Note that, not all instructions follow the aforementioned encoding principle of condition code and operands. So we may miss some rare instructions with special encoding patterns. However, according to the ARM manual [15], iDEV's refinement methods may only ignore five kinds of hint instructions with special purposes. These instructions are used to indicate the processor of a certain event(e.g., enter low-power state), and their execution results should be the same as *NOP* instruction without triggering any signals. Hence, it would not affect our empirical results even if not consider these hint instructions. We believe this preprocessing strategy achieves a reasonable trade-off.

## 4.3 Testing of Dynamic Execution

After getting the refined instruction set, iDEV performs an automated differential testing to examine the execution deviation (Type1 ISDev cases) among different hardware devices and emulators.

A straightforward method to recognize the runtime behavior of an instruction is feeding it into platforms and monitoring the output, as used in TaintInduce [12]. However, it is infeasible to record the impact of an instruction executed on hardware in this way. To do so, a huge amount of memory data has to be recorded and monitored, which makes testing and analysis storage-intensive and time-consuming. Therefore, in order to perform the testing effectively, we only audit a specific runtime behavior, i.e., Linux signal, to represent the runtime behaviors of instruction execution.

Instructions from the refined set are fetched one by one and executed by iDEV's testing engine. The testing engine initializes the device state before executing the instruction, then monitors the signal triggered by the instruction execution. To test a given instruction, the initial state will be set to the same on different platforms. Moreover, the testing engine will save and restore states before and after testing each instruction, in order to avoid potential exceptions caused by a sequence of instructions.

**Differential Analysis.** Executing an instruction could yield different signals, partially revealing the instruction's semantics. For instance, an instruction that triggers *SIGILL* signal is an illegal instruction, indicating it may be undefined in ARM ISA. And an instruction that triggers *SIGSEGV* will access memory, but fails during the testing. Note that, for the instructions that do not raise any signals which indicate normal execution, iDEV would mark their execution signal as *NONE*. If two platforms yield different signals when testing the same instruction under, iDEV could infer that this instruction could cause execution (Type1) deviation.

Therefore, iDEV simply compares the signals raised by instruction execution on different platforms to explore execution deviations, and records the deviation cases for further root cause analysis.

## 4.4 Testing of Static Disassembly

iDEV further performs differential testing on static analysis tools that could process ARM instructions. Disassembly is always the first step of binary analysis, hence disassemblers are chosen as targets to test, and iDEV collects deviations they yield when decoding ARM instructions. The workflow is similar to dynamic execution testing, but has an extra step of disassembly result normalization.

**Disassembly Results Normalization.** It is well known that different disassemblers could have different syntax formats. Such a difference could cause false positives when exploring semantic deviations. Therefore, iDEV has to normalize the outputs of disassembly and remove such noises when performing differential testing.

First, in the ARM architecture, the registers R9-R12 have alias names SB, SL, FP, IP respectively. Different disassembler developers would choose different register naming conventions. For example, Capstone [5] uses the alias names (SB-IP), while IDA Pro [40] chooses the original names (R9-R12). Second, ARM instructions may also have alias mnemonics. For instance, *LDM* has two alias mnemonics *LDMIA* and *LDMFD*, Objdump [45] uses *LDMFD*, while Capstone and Ghidra [35] use *LDM* and *LDMIA* respectively. Furthermore, there is no uniform standard on how to represent an immediate number in ARM instructions. Miasm [42] uses hexadecimal representation for constants, while Capstone uses decimal forms with a "#" prefix.

Therefore, before performing the differential testing, iDEV applies a tedious process to normalize the decoding results of each disassembler to ensure that all outputs come with the same format.

**Differential Analysis.** If all disassemblers have no objections to the decoding results of an instruction, then this instruction is likely to be correctly decoded by all of them, since the chance that different disassemblers make the same mistake is small. On the other hand, if disassemblers fail to reach an agreement on the disassembly result of an instruction, there must be at least one disassembler that has a wrong decoding result.

After normalizing the decoding results, iDEV also compares the decoding results of all disassemblers to collect instructions that could cause disassembly (Type2) deviations for further analysis.

## 4.5 Execution-Disassembly Cross-Checking

As we discussed in 3.1, not only within dynamic execution platforms and static disassembly tools, there are also semantic deviations between them. For example, instructions that can be executed should also be decodable, while instructions that trigger SIGILL signal during execution should not be decodable by disassemblers. If an instruction's execution signals and disassembly result do not match this correspondence, then there is an execution-disassembly (Type3) deviation, which may also bring potential implications.

Therefore, iDEV further cross-checks the results of static disassembly and dynamic execution, and records instructions that cause Type3 deviations for the following root cause analysis.

# 5 EVALUATION

In this section, we elaborate the evaluation of iDEV. We performed several experiments to answer the following research questions:

- **RQ1: Efficiency of iDEV's preprocessing:** How efficient is iDEV to explore the preprocessed ARM instructions set?
- **RQ2: Prevalence of semantic deviations:** How prevalent are semantic deviations in ARM instruction processing?
- **RQ3: Root causes of semantic deviations:** What are the root causes of these deviations?

## 5.1 Experiment Setup

We selected four hardware devices including Tenda AC9 router, ASUS RT-AC68U router, Raspberry Pi 2B and Pi 3B+, and a virtual device QEMU-arm-system-2.11.1 raspi2 as the dynamic execution platforms to test. We chose these physical devices because they were commonly used so that their toolchains could be obtained easily. Besides, their CPUs covered three different specific implementations (including cortex-A7/A9/A53) of ARM32 ISA, which could allow us to better explore the deviations of the same instruction set under different implementations. It should be noted that the AArch32 ISA used by Pi 3B+ is fully backward compatible with the ARMv7A, and the choice of this device also made us observe the evolution of ISA to a certain extent, which we would demonstrate in Section 6. In addition, we chose to test five most popular disassemblers including Capstone-4.0.1 [5], Objdump-2.26.1 [45], IDA Pro-7.5.200619 [40], Ghidra-9.1.1 [35], and Miasm-0.1.3 [42].

iDEV's execution testing engine was implemented in C with about 500 lines of code, and its disassembly testing and differential analysis scripts were implemented in Python with 3.5K lines of code. During the evaluation, we found some targets (e.g., Tenda AC9) do not have sufficient resources to finish dynamic testing all at once, and thus divided the testing tasks into multiple batches.

**Table 1: The throughput and testing time with or without refinement (ref.) during dynamic execution testing.**

| Platform | CPU | Average Throughput | Testing Time | | File Size |
|---|---|---|---|---|---|
| | | | w/ ref. | w/o ref. | |
| **Pi 2B** | Cortex-A7 | 4,923 Inst/s | > 240h | 114 min | |
| **Pi 3B+** | Cortex-A53 | 9,834 Inst/s | > 120h | 57 min | |
| **Tenda AC9** | Cortex-A7 | 6,096 Inst/s | >190h | 192 min | 600MB |
| **RT-AC68U** | Cortex-A9 | 9,283 Inst/s | > 110h | 60 min | |
| **QEMU-Pi 2B** | N/A | 988 Inst/s | > 1200h | 566 min | |

In addition, we developed a series of auxiliary programs to help manually classify ISDev cases and analyze root causes.

## 5.2 Effectiveness of Preprocessing (RQ1)

As aforementioned, we preprocessed the ARM instruction set to reduce the scale. To illustrate the effectiveness of preprocessing, we measured the testing efficiency of dynamic execution testing.

Table 1 shows the throughput and testing time of dynamic execution testing as well as the file size for the refined instructions with signal information. The throughput is calculated by testing 1 million instructions on each platform three times. According to the result, Pi 3B+ has the highest throughput which is up to 9,834 instructions per second. QEMU-emulated Pi 2B, on the contrary, has the lowest throughput which is 988 instructions per second. This is reasonable since TCG translation used by QEMU for ARM instruction emulation inevitably brings large overhead.

Under such throughput rates, it is hard to perform full testing of all ARM instructions on these devices due to their small memory and weak computing capabilities. The testing time before refinement is an estimated time based on the throughput, which would take hundreds of hours and it is unacceptable for devices with low performance. Besides, the dynamic execution testing engine feeds arbitrary instructions to target platforms to execute, and will inevitably crash in kinds of corner cases. Thus, it requires some human interventions to keep devices online.

But after the refinements, the number of instructions under test is reduced by several orders of magnitude. So, we could finish a round of tests within 10 hours on each platform. During the analysis phase, we manually analyzed for root causes of the ISDev cases. Even auxiliary scripts do help to some extent, we still had to carefully synchronize our results with ARM specification, which had taken several weeks to analyze millions of instructions with semantic deviations in the refined instruction set.

In addition, the refinement also helps mitigate the load of storage of instructions with attached information. Embedded devices in general have small storage. For example, our refined results occupy about 600MB for $2^{25}$ instructions, while Tenda AC9 only has 64MB DRAM storage. In this case, even after refinement, we have to split the whole testing into multiple parts. Therefore, the instruction set preprocessing is necessary for our work.

## 5.3 Deviation Prevalence Evaluation (RQ2)

*5.3.1 Type1 Deviation Result.* Through the initial testing result, we first found that all instructions starting with *0xef* have the same execution behaviors on each platform. After referring to the ARM manual, we found that these instructions are *SVC* instructions. As the manual says [15], *software can use this instruction as a call to an operating system to provide a service.* Thus, the execution result of *SVC* is determined by the operating system rather than the platform

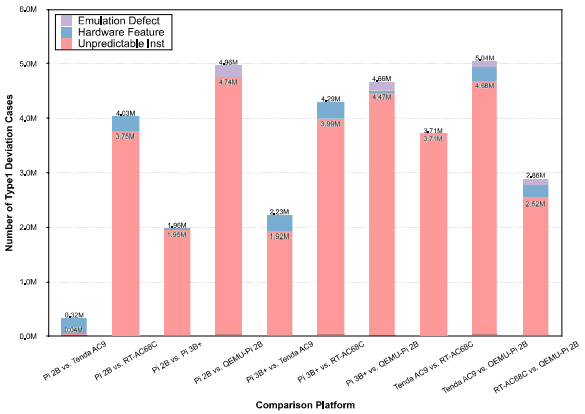



**Figure 5: Number of Type1 deviation cases between any two platforms with root causes proportion.**

itself, then we removed this part of data from our result. Figure 5 shows the Type1 deviation cases between any two devices, and the colors represent the proportion of each root cause.

After removing *SVC*, we found that the number of Type1 deviations between Pi 2B and Tenda AC9 is the lowest, which is about 323 thousand. On the other hand, the deviations between QEMU-Pi 2B and Tenda AC9 are the most, which is over 5 million. For all any other platform pairs, there are over 1 million instructions which could cause Type1 deviations between them. Besides, we counted the total number of instructions that would cause Type1 deviations in at least one pair of platforms. After removing *SVC*, a total of **6,355,355** instructions are reported. Note that, the total number of instructions that have been tested is $2^{25}$. Therefore, over **18.9%** of instructions could cause Type1 deviations in different devices.

It is worth noting that QEMU-Pi 2B is the main platform that causes Type1 deviation in instruction execution. Any platform pairs including QEMU have more than 4 million deviation instructions. This is mainly because QEMU tends to emulate unpredictable instructions as executable, which leads to many subtle differences between QEMU and physical hardware.

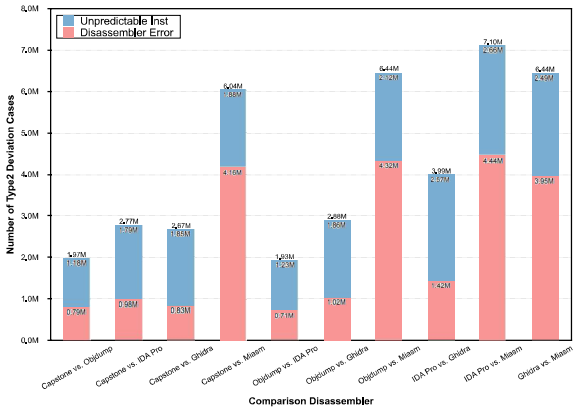



**Figure 6: Number of Type2 deviation cases between any two disassembles with root causes proportion.**

In general, this result shows that the deviation in the execution of instructions is prevalent among different platforms.

*5.3.2 Type2 Deviation Result.* Similarly, we also calculated the number of instructions that could cause deviations between each pair of disassemblers, and the total number of instructions that cause deviations in at least one pair of disassemblers. Figure 6 displayed the number of Type2 deviation cases between any two disassembles with the proportion of root causes.

As we can see, for each pair of disassemblers, the number of instructions with disassembly deviations is more than 1 million. In particular, there are even more than 7 million instructions that would cause deviations between IDA Pro and Miasm.

In total, there are **8,815,368** instructions which can cause deviations in at least one pair of disassemblers, accounting for **26.27%** of all instructions that have been tested.

*5.3.3 Type3 Deviation Result.* Finally, we measured the Type3 deviation between execution platforms and disassembles. As aforementioned, there are two sub-types of deviations, i.e., Type3-1 (executable-undecodable) and Type3-2 (unexecutable-decodable). Figure 7 shows the heatmap of each types of deviations. The lightness of the color in the heatmap indicates the Type3 deviation of the corresponding platform-disassembler pair.

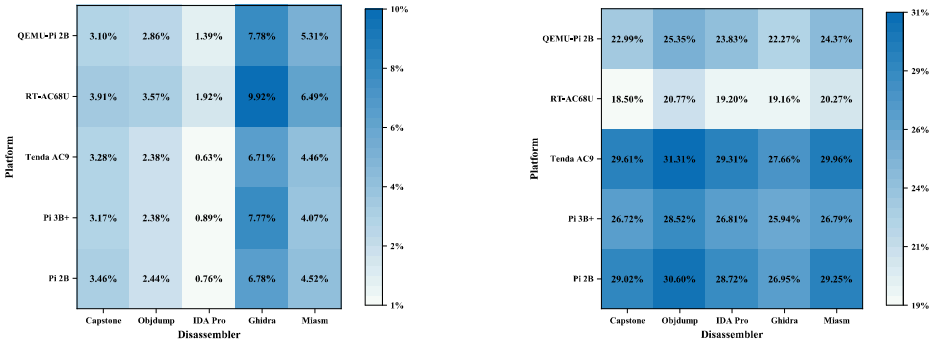
We could find that different platform-disassembler pairs have a different degree of Type3 deviation. Specifically, the pairs involving Ghidra have the largest number of executable-undecodable instructions, implying that attackers could easily stop binary analysis solutions based on Ghidra. Especially for the combination of RT-AC68U and Ghidra, which has more than 3.3 million Type3-1 deviation instructions. In addition, Objdump would cause the most Type3-2 `ISDev` cases, and the pair of Tenda AC9 and Objdump has more than 10 million unexecutable-decodable instructions which is the largest number among all pairs. Hence, attackers could have large chances to hide malicious code from analysis.

Note that, Type3-1 deviation instructions in any pair account for over 0.5% of all instructions that have been tested, while Type3-2 deviation in each pair accounts for over 18% of all instructions that have been tested. The latter ratio is higher, indicating that disassemblers always tend to perform decoding even if instructions may not executable (e.g., unpredictable).

## 5.4 Root Cause Evaluation (RQ3)

*5.4.1 Analysis on Type1 Deviation.* First, we analyzed instructions that cause deviations in execution platforms. After comparing the runtime signals and verifying with the ARM manual, we found that the Type1 deviations are mainly caused by the following reasons:

- *Hardware Features Support:* CPUs of the same architecture made by different vendors could support different features defined by the ISA. For instance, some special extensions of the instruction set are not supported in all hardware. Specifically, the Tenda AC9 hardware does not have vector operation support (VFP), thus can not execute floating-point instructions.
- *Unpredictable Instructions:* As aforementioned, the ARM manual states that unpredictable instructions could have undefined behaviors. Each CPU could have its own implementation, which may be different from others. Hence, unpredictable instructions inevitably have semantics deviation. Taking the instruction *LDR*




**(a) Heatmap of Type3-1 deviation cases** **(b) Heatmap of Type3-2 deviation cases**

**Figure 7: Heatmap of instruction with Type3 deviation.**

as an example, if the *Rn* and *Rt* fields of the instruction encoding are the same, then this instruction is unpredictable. In practice, the Pi 2B hardware will yield a SIGILL signal when executing this instruction, while a Pi 3B+ device will not.

- *Emulation Defect:* Emulation tools (e.g., QEMU) could introduce bugs in their software implementations due to human errors, or just cannot keep pace with the latest ISA specification due to manual resource limitation.
- *The SVC Instruction:* As we discussed in 5.3.1, Semantic deviations of these *SVC* instructions are not caused by hardware or emulators, thus were removed from further analysis.

Figure 5 has shown the proportion of different root causes for Type1 deviation (without *SVC*) between any two platforms.

The pair of Pi 2B and Tenda AC9 has the lowest number of instructions with Type1 deviation, and over 87.45% of them are mainly due to different hardware feature supports provided by these two devices. Specifically, Tenda AC9 router does not support hardware features including VFP, which accounts for the most deviations comparing to Pi 2B.

For the pair of Tenda AC9 and QEMU-Pi 2B, which has the largest number of Type1 deviation instructions, about 90% of their deviations are caused by unpredictable instructions. It means that different devices indeed process unpredictable instructions very differently. And as we mentioned before, QEMU tends to emulate unpredictable instruction as executable.

For the pair of Pi 2B and Pi 3B+ which have a small number of deviations, unpredictable instructions account for over 99% of deviations. It further confirms that different devices tend to process unpredictable instructions arbitrarily.

*5.4.2 Analysis on Type2 Deviation.* Second, we analyzed instructions that cause deviations to binary analysis tools, especially disassemblers. After comparing disassembly results and verifying with the ARM manual, we found that deviations in ARM instruction decoding or disassembly are attributed to two reasons: *unpredictable instructions* and *disassembler errors*.

For unpredictable instructions, the ARM manual states that their behaviors are undefined. Therefore, different disassemblers could have different choices to decode such instructions, inevitably causing deviations similar to dynamic execution platforms.

For other instructions, disassemblers may have errors and thus have divergences with the architecture ISA. Such disassembler errors can be classified into the following three categories: (1) *Under decoding*. It represents errors that disassemblers fail to recognize or decode valid instructions. (2) *Incorrect decoding*. It represents errors that disassemblers incorrectly decode valid instructions. (3) *Over decoding:* A disassembler may *successfully* decode instructions which are explicitly marked as UNDEFINED in ARM manual.

In addition to the above classification criteria, there is a special case. The *UDF* instruction with encoding of `0xe7fedef0` is *defined as permanently UNDEFINED* according to ARM specification [15]. But Capstone [5] decodes this special encoding as *TRAP*, while normally decoding *UDF* instructions of other encodings.

As shown in Figure 6, only a part of Type2 `ISDev` cases between disassemblers are caused by unpredictable instructions. As discussed in Section 5.4.2, deviations caused by disassembly errors could also bring security implications. We thus further studied the distribution of these disassembly errors, as summarized in Figure 8.

Results showed that, among these five disassemblers, Capstone performs the best to disassemble the ARMv7-A instruction set. It only makes mistakes for two instruction mnemonics. Specifically, these errors are not semantic errors, but errors on conditional execution codes. Among them, some *MSR* instructions and *MRS* instructions are decoded as *MSREQ* and *MRSEQ* respectively.

We found Miasm does the worst in instruction disassembly. It under-decodes more than 174 thousand instructions, and incorrectly decodes more than 2 million encodings. Since Miasm is a complete reverse engineering framework rather than a simple disassembler,

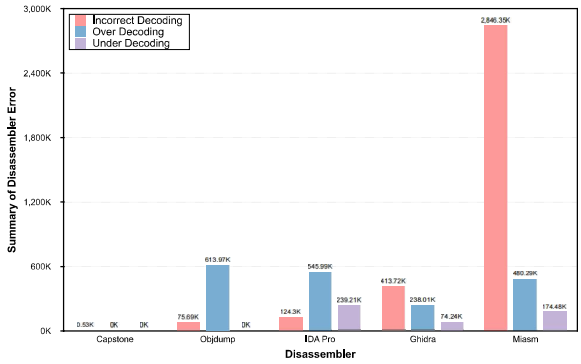



**Figure 8: Summary of disassembler errors.**

**Table 2: Proportion of unpredictable instruction in instructions causing execution-disassembly deviations.**

| Platform | Proportion of Unpredictable Instruction for Execution-Disassembly Deviation (Executable-Undecodable) | | | | |
|---|---|---|---|---|---|
| | Capstone | Objdump | IDA Pro | Ghidra | Miasm |
| Pi 2B | 81.09% | 87.77% | 83.73% | 87.01% | 71.68% |
| Pi 3B+ | 88.96% | 87.05% | 83.80% | 78.94% | 76.87% |
| Tenda AC9 | 81.25% | 88.89% | 85.49% | 87.40% | 72.10% |
| RT-AC68U | 81.34% | 91.20% | 80.91% | 78.96% | 78.99% |
| QEMU-Pi 2B | 90.43% | 95.49% | 90.17% | 94.98% | 86.60% |
| **Platform** | **Proportion of Unpredictable Instruction for Execution-Disassembly Deviation (Unexecutable-Decodable)** | | | | |
| | Capstone | Objdump | IDA Pro | Ghidra | Miasm |
| Pi 2B | 36.11% | 32.76% | 33.91% | 32.50% | 31.51% |
| Pi 3B+ | 32.91% | 28.93% | 30.29% | 28.14% | 27.64% |
| Tenda AC9 | 35.34% | 32.34% | 33.19% | 32.03% | 31.10% |
| RT-AC68U | 7.23% | 7.84% | 6.71% | 6.18% | 7.88% |
| QEMU-Pi 2B | 4.55% | 4.68% | 4.57% | 3.74% | 4.70% |

these disassembly errors could further cause more errors to its high-level analysis processes, such as binary lifting and CFG recovery.

*5.4.3 Analysis on Type3 Deviation.* As aforementioned, the instruction deviation between execution and disassembly could be divided into two specific scenarios, hence the root causes of them are also different.

**Type3-1: Executable-Undecodable.** For instructions that are executable on a platform but cannot be decoded by a disassembler, the root causes can be classified into three categories: (1) *Unpredictable instruction.* The instruction is unpredictable, i.e., not clearly defined in the ISA. (2) *Under decoding.* The instruction is valid but the disassembler fails to recognize it. (3) *Emulation defect.* The instruction is not defined but the emulator could execute it.

**Type3-2: Decodable-Unexecutable.** For instructions that could be decoded by a disassembler but cannot be executed on a platform, the root causes can be classified into four categories: (1) *Unpredictable instruction.* The instruction is unpredictable. (2) *Over decoding.* The instruction is invalid but the disassembler wrongly decodes it. (3) *Hardware feature support.* The instruction can only be executed on devices that have some specific hardware features. (4) *Emulation defect.* The instruction is valid but the (virtual) device fails to execute it.

For instructions that cause deviations to execution-disassembly platforms, we calculated the ratio of ones that are caused by unpredictable instructions. For ones that are caused by disassembler errors or emulation defects, they could be patched by updating software. Unpredictable instructions cannot be easily fixed due to the inherent ambiguity of their definitions in ARM ISA. The ratio of unpredictable instructions is listed in Table 2.

It shows that unpredictable instructions account for over 80% of Type3-1 deviation instructions, except in some pairs related to Miasm, which is because that Miasm has many under decoding errors that accounts for a large portion of deviations.

As for Type3-2 deviation instructions, the proportion of unpredictable instructions is around 4% to 35%. Such a difference is mainly caused by hardware features. Hardware manufacturers tend to support a limited number of hardware features that are sufficient for application scenarios. On the other hand, disassemblers tend to decode instructions as much as possible. A special case here is the QEMU emulator, which has the lowest proportion of unpredictable instructions. As an emulator, QEMU inherently prefers to emulate unpredictable instructions as executable.

# 6 LESSONS LEARNED AND INSIGHTS

## 6.1 Lessons on Unpredictable Instruction

Based on the empirical study results, we find that unpredictable instructions account for a large proportion of `ISDev` issues. We thus further conducted an in-depth analysis of these instructions, and have learned the following lessons.

**Hardware Resource Consideration.** Physical CPUs in general process unpredictable instructions in two ways: (1) refuse to execute them and yield a SIGILL signal, or (2) execute them as normal. By monitoring register values before and after the instruction execution, we confirmed that, unpredictable instructions that are executable in general have the same functionalities as the standard instructions they derived from. We speculate that this behavior may be due to the consideration of saving circuits during the chip design. The CPU may skip certain checks on unpredictable instruction conditions, and simply execute unpredictable instructions as standard instructions.

**Software Developer Variances.** We analyzed the source code of the QEMU emulator and found out it does not have a unified way to handle unpredictable instructions. For some instructions (e.g., *ROR*), QEMU does not check the unpredictable condition bits, and translates both standard and unpredictable instructions into the same tiny code generator (TCG). For some other instructions (e.g., *REV*), QEMU would decode instructions that meet the unpredictable condition into *illegal_op*, thus yields a SIGILL signal during execution. We infer that, this inconsistency could be due to the fact that the codes are developed by different developers.

As for deviations in disassemblers, we have observed a similar phenomenon as in QEMU, i.e., each disassembler does not have a unified way to process unpredictable instructions. Take the disassembler Capstone [5] as an example. After analyzing its source code of decoding ARM32 instructions, we find that, (1) for some instructions (e.g., *LSL*), Capstone would treat the unpredictable condition bits as a part of the decoding rules, thus fail to decode such non-standard unpredictable instructions; (2) for some other instructions (e.g., *MUL*), Capstone ignores the check of these unpredictable condition bits and decodes them as the standard instructions they derived from.

**ISA Evolution.** Unpredictable condition bits in instructions are often used as reserved bits, which are reserved for future instruction encoding. In this way, it could leave room for future ISA upgrades while providing flexibility to vendors at the same time. For instance, in the ARMv7-A ISA, bits[8:11] of the *STREX* instruction encoding are all unpredictable condition bits. Unpredictable instructions derived from it can be executed on Pi 2B and other ARMv7-A devices in our study, same as the standard *STREX* instruction. But in the ARMv8 ISA's AArch32 instruction set, which is backward compatible with the ARMv7-A instruction set, bits[8:9] in the *STREX* instruction encoding are fixated as 1, and other bit representations yield different instructions.

**The Unavoidability of ISDev Issue.** The ambiguous definition of unpredictable instructions in the ARM specification is the major root cause of `ISDev`. It enables execution platforms and binary

analysis tools (e.g., disassemblers) to take different actions when processing such instructions, and leads to a considerable amount of semantic deviations. Even in an individual platform or tool, it may take non-unified actions when processing unpredictable instructions, since developers are different. As long as the ARM specification is ambiguous, this ISDev issue will always exist.

### 6.2 Mitigations to ISDev Implications

As we mentioned before, the ISDev issue may bring security implications, and we have demonstrated several proof-of-concept attacks already. To mitigate such implications, the most straightforward solution is to eliminate instruction semantic ambiguity introduced in the ARM manual. For example, if all CPU manufacturers implement unpredictable instructions as undefined instructions, and all binary analysis tools never process these instructions, then threats caused by unpredictable instructions could be defeated. Meanwhile, binary analysis tools could utilize differential testing to locate programming errors, and mitigate the rest of the deviations as well as threats. However, this solution may be unrealistic, since manufacturers have the right to implement specific semantics for unpredictable instructions as allowed in the license.

An alternative way is updating binary analysis tools to recognize unpredictable instructions and alert users. Objdump has set up a good example of dealing with unpredictable instructions. Specifically, it decodes such instructions as to their normal semantics, but attaches an annotation of *<UNPREDICTABLE>* to warn users. So, analysts or other tools can pay more attention to the behaviors of these instructions. However, this solution could only mitigate semantic deviation threats to a limited extent. According to our testing, Objdump fails to recognize many unpredictable instructions as well. Moreover, users of these binary analysis tools have to understand the attached information and carefully perform the following actions, which is tedious and error-prone.

## 7 DISCUSSION

### 7.1 Threats to Validity

In our experiments, we have identified the threats to validity are mainly lying in the following aspects.

**Selection of Evaluation Targets.** In this empirical study, the choice of evaluation targets may bias the results. To mitigate this threat, we tried to make the testing targets more diverse when performing the evaluation. For instruction execution platforms, although our research target is the ARMv7-A ISA, we chose to use different specific ARM CPUs under this instruction set architecture as our evaluation devices. In addition, we also used a QEMU-based virtual device to further increase the diversity of our evaluation targets. Regarding disassembly, we ensured that the selected disassemblers use different disassembly engines, which could further prove the prevalence of the ISDev issue. Future work could apply iDEV to the testing on other different ARM CPUs and disassemblers, to obtain more comprehensive empirical results.

**Instruction Execution Context.** Another threat is the device context (e.g., register values) may affect the execution result of instruction among different platforms, while this research should only consider the execution deviation brought by different CPUs. To reduce this threat, iDEV saves the register values except for *PC*, *SP* and clears them before executing the instruction under test, then restores the register values after the instruction execution to avoid deviation false positives as much as possible. For a few instructions that crash the execution testing engine due to corrupting the SP/PC register, we have constructed a blacklist to shield the impact of these instructions. But in fact, the crash of the testing engine also suggests that these instructions are not undefined, and they are successfully executed by the CPU instead, which would not affect our evaluation results.

**Internal Validity.** The major threat to internal validity lies in the possible bugs in our implementation of iDEV and the errors in the manual analysis phase. To increase the confidence of this study, we have peer-reviewed our code, and repeatedly analyzed the testing results. In addition, we have further manually verified some ISDev cases on different platforms or disassemblers, and ensure that these cases could indeed yield deviation results on different instruction processing entities. We plan to make iDEV open source to enable the research community to build on top of our work in addressing ISDev issues.

### 7.2 Limitations

This study has some other limitations. First, our iDEV prototype only concentrates on the ARMv7-A (AArch32) 32-bit instruction set, without considering the Thumb instruction set and the 64-bit architecture (i.e., AArch64). But the ISDev problem is not ARM32-specific, and neither are techniques used in iDEV. For instance, in the ARMv8 ISA specification [16], the definition of unpredictable instruction is still ambiguous on AArch64 architecture. Thus, we believe iDEV could be transported to other platforms easily.

Second, iDEV framework may have false negatives. For the differential testing engine, if all 5 disassemblers have an agreement on decoding one instruction, then iDEV will not doubt it, even if all disassemblers have made a mistake. This is an intrinsic problem for all differential testing based solutions. As for the execution deviation, iDEV can only reveal a subset of potential deviation by only considering the signal divergences. Despite the potential false negatives, iDEV has successfully revealed the prevalence of deviations in ARM ISA which may involve security issues, as expected.

Third, our iDEV prototype is not fully automated yet. Since there is no standard or reference implementation, we can not rely on the decoding results of any disassembler, nor the instruction execution results of any execution platform. Therefore, many human efforts are needed to analyze the deviations.

## 8 RELATED WORKS

**ISDev Issues.** Some previous works [19, 24, 32, 38, 41] have explored the semantic deviation of binary instructions by using differential testing. Emufuzzer [32] and PROTEUS [41] studied the execution deviation (Type1) of instructions between the emulator and physical CPU to discover emulator bugs and to perform emulator evasion. N-version disassembly [38] and Fleece [24] located disassembler bugs by exploring the disassembly deviation (Type2) of instructions among different disassemblers. Sandsifter [19] dug hidden instructions and software vulnerabilities by comparing the consistency deviation between instruction execution and disassembly (Type3). Besides, only a few of them systematically studied the

**Table 3: A summary of the works on instruction deviation using differential testing. √* means partial support.**

| | Target Instruction Set Architecture | Explored Instruction | | Semantic Deviation Type | | | Security Implication Analysis |
|---|---|---|---|---|---|---|---|
| | | Type1 | Type2 | Type3 | | | |
| | | | | Type3-1 | Type3-2 | | |
| **EmuFuzzer** | x86 | √* | × | × | × | | √* |
| **N-version disassembly** | x86 | × | √ | × | × | | √* |
| **Fleece** | x86/PowerPC/ARM | × | √ | × | × | | √* |
| **Sandsifter** | x86 | × | × | √ | × | | √* |
| **PROTEUS** | ARM | √* | × | × | × | | √* |
| **iDEV** | ARM | √ | √ | √ | √ | | √ |

security impact of such issues. Although PROTEUS [41] presented that malware could exploit ARM instruction deviations between CPU and emulators to detect virtual environments, it not treated unpredictable instruction as a reliable exploit. However, we found the deviations in these instructions are still exploitable for adversaries. Our `iDEV` framework has systematically explored all kinds of semantic deviations and has revealed the security implications from a more general perspective.

Table 3 summarizes the target instruction set architecture and the types of instruction semantic deviation studied in these works, where √* means that Emufuzzer and PROTEUS have only considered the execution deviation between the emulator and the physical CPU, but not the deviations among different physical CPUs. In general, `iDEV` characterized the instruction semantic deviation (`ISDev`) issue and has conducted a comprehensive empirical study on ARM ISA for the first time.

**CPU Threats.** Apart from the instruction semantic deviation issue, many hardware vulnerabilities including CPU side channels have been exposed in the past decade, drawing increasing attention from the community. With the measurement technique of PRIME+PROBE [31] and FLUSH+RELOAD [50], architectural features like out-of-order execution and branch prediction, are exploited to probe secrete data in CPU internal cache, which are well-known as Meltdown [30], Spectre [27], Foreshadow [8, 48] and so on. Besides, attackers also designed side-channel attacks with Microarchitectural Data Sampling (MDS) to access unauthorized sensitive information [9, 46]. To mitigate these issues, there are various mitigating solutions put forward from both hardware and software [21–23]. However, most of them bring extreme overhead to CPU performance.

In addition to the design flaws within CPU microarchitecture, there are latent security issues in CPU instruction sets. In 1994, the Intel Pentium series CPU was discovered to have bugs in their floating point unit (FPU) [13], which might result in an incorrect calculation when executing dividing instructions (*FDIV*). In 1997, another *F00F* bug [14] in Pentium instruction set was revealed. When the instruction `lock cmpxchg8b eax` (encoded as `0xf00fc7c8`) was executed, it locks down the processor until a physical reboot. In 2018, Christopher Domas first presented the possibility for exploiting the hidden instructions of x86 ISA to implant backdoors in real-world CPU [20]. UISFuzz [29] optimized the fuzzing efficiency to explore undocumented instructions. But these works only concentrate on the hidden instructions of x86 instruction set architecture.

**Differential Analysis.** As early as 1998, Bellare et al. proposed the method of using differential testing for software testing [33]. In addition to test emulator and disassembly tools we mentioned before, differential testing is also widely used on the testing of various types of software, including C compilers [28, 44, 49], JVMs implementations [10, 11], program analyzers [26], SSL/TLS certification validation [6, 39, 43, 47]. Besides, MeanDiff [25] used symbolic equivalence check in combination with differential testing to test the correctness of IR semantics, and Nilizadeh et al. [34] used differential fuzzing for side-channel attacks detection.

## 9 CONCLUSION

In this paper, we summarized the instruction semantic deviation and proposed `ISDev` problem which could bring security implications in ARM instruction processing. Then we developed `iDEV`, a framework to systematically explore `ISDev` issue of ARM instruction with differential testing. We evaluated `iDEV` on four physical CPUs, the QEMU emulator, and five disassemblers that could process the ARMv7-A instruction set, and revealed the prevalence and root cause of these instruction deviations.

## ACKNOWLEDGEMENT

This work was supported in part by National Natural Science Foundation of China under Grant U1736209, 61972224 and 61772308, and BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009.

## REFERENCES

[1] 2021. Dyninst. https://github.com/dyninst/dyninst.
[2] 2021. LLVM. https://llvm.org/.
[3] 2021. QEMU. https://www.qemu.org/.
[4] 2021. Valgrind. http://valgrind.org/.
[5] Nguyen Anh Quynh. 2020. Capstone. https://github.com/aquynh/capstone.
[6] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *2014 IEEE Symposium on Security and Privacy*. 114–129. https://doi.org/10.1109/SP.2014.15
[7] Martin Brunner, Hans Hofinger, Christoph Krauß, Christopher Roblee, P Schoo, and S Todt. 2010. Infiltrating critical infrastructures with next-generation attacks. *Fraunhofer Institute for Secure Information Technology (SIT), Munich* (2010).
[8] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck
[9] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-Resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 769–784. https://doi.org/10.1145/3319535.3363219
[10] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1257–1268. https://doi.org/10.1109/ICSE.2019.00127
[11] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 85–99. https://doi.org/10.1145/2908080.2908095
[12] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. 2019. One Engine To Serve'em All: Inferring Taint Rules Without Architectural Semantics.. In *NDSS*. https://doi.org/10.14722/ndss.2019.23339
[13] Tim Coe. 1995. Inside the pentium-fdiv bug. *DR DOBBS JOURNAL* 20, 4 (1995), 129.
[14] Robert R Collins. 1997. The intel pentium f00f bug description and workarounds. *Doctor Dobb's Journal* (1997).

[15] ARM Corporation. 2018. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition.* ARM Corporation.

[16] ARM Corporation. 2018. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile.* ARM Corporation.

[17] Intel Corporation. [n.d.]. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[18] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. 472–489. https://doi.org/10.1109/SP.2019.00003

[19] Christopher Domas. 2017. Breaking the x86 ISA. *Black Hat* (2017).

[20] Christopher Domas. 2018. Hardware Backdoors in x86 CPUs. *Black Hat* (2018).

[21] Brendan Gregg. 2018. KPTI/KAISER meltdown initial performance regressions.

[22] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*. Springer, 161–176. https://doi.org/10.1007/978-3-319-62105-0_11

[23] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. 2018. EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 255–266. https://www.usenix.org/conference/atc18/presentation/hua

[24] Nathan Jay and Barton P. Miller. 2018. Structured random differential testing of instruction decoders. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 84–94. https://doi.org/10.1109/SANER.2018.8330199

[25] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungll Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing intermediate representations for binary analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 353–364. https://doi.org/10.1109/ASE.2017.8115648

[26] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/3293882.3330553

[27] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP.2019.00002

[28] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

[29] Xixing Li, Zehui Wu, Qiang Wei, and Haolan Wu. 2019. UISFuzz: An Efficient Fuzzing Method for CPU Undocumented Instruction Searching. *IEEE Access* 7 (2019), 149224–149236. https://doi.org/10.1109/ACCESS.2019.2946444

[30] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[31] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622. https://doi.org/10.1109/SP.2015.43

[32] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/1572272.1572303

[33] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[34] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. 2019. DiffFuzz: Differential Fuzzing for Side-Channel Analysis. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 176–187. https://doi.org/10.1109/ICSE.2019.00034

[35] NSA. 2021. Ghidra. https://github.com/NationalSecurityAgency/ghidra.

[36] Trail of Bits. 2021. McSema. https://github.com/lifting-bits/mcsema.

[37] Trail of Bits. 2021. mishegos. https://github.com/trailofbits/mishegos.

[38] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. N-Version Disassembly: Differential Testing of X86 Disassemblers. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 265–274. https://doi.org/10.1145/1831708.1831741

[39] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 615–632. https://doi.org/10.1109/SP.2017.27

[40] Hex Rays. 2021. IDA Pro. https://www.hex-rays.com/products/ida/.

[41] Onur Sahin, Ayse K Coskun, and Manuel Egele. 2018. Proteus: Detecting Android Emulators from Instruction-Level Profiles. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 3–24. https://doi.org/10.1007/978-3-030-00470-5_1

[42] CEA IT Security. 2021. Miasm. https://github.com/cea-sec/miasm.

[43] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. 2017. HVLearn: Automated Black-Box Analysis of Hostname Verification in SSL/TLS Implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*. 521–538. https://doi.org/10.1109/SP.2017.46

[44] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 203–213. https://doi.org/10.1145/2884781.2884879

[45] GNU Binary Utilities. 2021. Objdump. https://www.gnu.org/software/binutils/.

[46] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *2019 IEEE Symposium on Security and Privacy (SP)*. 88–105. https://doi.org/10.1109/SP.2019.00087

[47] Andreas Walz and Axel Sikora. 2020. Exploiting Dissent: Towards Fuzzing-Based Differential Black-Box Testing of TLS Implementations. *IEEE Transactions on Dependable and Secure Computing* 17, 2 (2020), 278–291. https://doi.org/10.1109/TDSC.2017.2763947

[48] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018).

[49] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[50] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom