Noname manuscript No.
(will be inserted by the editor)

Link-time Smart Card Code Hardening

Ronald De Keulenaer · Jonas Maebe · Koen De Bosschere · Bjorn De Sutter

the date of receipt and acceptance should be inserted later

Keywords smart card, fault, software protection, binary rewriting, overhead

Abstract This paper presents a feasibility study to protect smart card software against fault-injection attacks by means of link-time code rewriting. This approach avoids the drawbacks of source code hardening, avoids the need for manual assembly writing, and is applicable in conjunction with closed third-party compilers. We implemented a range of cookbook code hardening recipes in a prototype link-time rewriter and evaluate their coverage and associated overhead to conclude that this approach is promising. We demonstrate that the overhead of using an automated link-time approach is not significantly higher than what can be obtained with compile-time hardening or with manual hardening of compiler-generated assembly code.

1 Introduction

Cryptographic keys and PIN hashes are often embedded in programmable hardware such as smart cards. To steal that data, attackers apply sophisticated attacks. During passive attacks, attackers observe the behavior of the chip executing code to reconstruct the program structure and obtain knowledge on its internal data values. The observable behavior includes timing, power consumption [36], electromagnetic radiation, etc. In active attacks, which often build on passive attacks, attackers intervene in the execution by injecting faults by means of power glitches, clock period alterations, temperature rises, active probing of buses, or light attacks [8]. Faults cause bit flips that can alter data values or program code, e.g., when an instruction being fetched is altered by a laser flash. When this remains undetected, security barriers risk being broken: private keys can leak when both correct and incorrect outputs are available [6]; private data can get exported because array bounds checks are corrupted [8]; skipping encryption rounds can leak keys [13]; and checks to prevent the output of invalid data can get circumvented [25].

To protect the software, hardware can provide redundancy, e.g., by using error-correcting coding [23], or redundancy can be inserted in the software to detect occurring faults [46]. The latter approach typically offers more flexibility with respect to the security policies that can be deployed in the field. To exploit the flexibility maximally, some fully automated compiler or code generation tool ideally can apply generic forms of low-overhead redundancy to implement security policies specified in a convenient form, i.e., without impeding the programmer's productivity.

This implies that programmers should not have to waste effort and time on changing their source code to introduce the redundancy, nor should they have to rewrite or even inspect the compiler-generated assembly code. Instead they should have to care only about the functional correctness of their source code.

Many redundancy schemes have been proposed to protect against single-event upsets (i.e, soft errors) [2, 7, 41–43, 39], to protect against targeted attacks [44, 24], and to prevent control flow from deviating from predetermined paths [1]. Automated support for these schemes, that also tries to limit the performance overhead of the introduced redundancy, is typically implemented in (research) compilers.

Authors addresses: R. De Keulenaer, J. Maebe, K. De Bosschere, B. De Sutter (corresponding author), Computer Systems Lab, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium. E-mail: bjorn.desutter@elis.ugent.be. This research is supported by the FWO project G.0198.09N. We also thank ARM for giving access to their compilers and tools.

In practice, however, companies rely on multiple in-house and third-party development tool chains that may change over time. To maintain interoperability with different tool chains and avoid vendor lock-in, tools that automate the implementation of security policies should therefore be separate tools that do not break existing tool chains and do not depend on the internal operation of the used compilers. This leaves two basic options to insert redundancy: source code rewriting and (post-) link-time binary code rewriting.

This paper presents a feasibility study of link-time binary code rewriting to protect against fault-injection attacks, assessing its impact on efficiency and effectiveness. The paper's contribution is its argumentation as to why link-time code rewriting is feasible with closed, third-party compiler tool flows, as well as an evaluation of the coverage and overhead obtained with a prototype link-time rewriter that implements four standard protections. Concretely, our experiments demonstrate that the overhead of this approach is not unacceptably higher than what could be achieved with other approaches, such as compile-time protection or manual assembly code protection.

We evaluate four cookbook recipes for local hardening of code against certain classes of single-instruction failures, i.e., single instructions that are skipped as the result of an injected fault [37]. With these protections, we target the ARM Cortex-M0, the core used in ARM smart card SecurCore SC000 processors [52].

We know of no automated fault-injection protection tools in use today for smart card software. The resulting need to provide this protection manually is one of the main reasons why assembly programming is still so common in this domain. By providing convincing arguments for automating this protection in a tool that does not disrupt proprietary compiler tool flows and that introduces only a minimal amount of additional overhead, we hope to contribute a significant step towards more productive smart card programming.

This paper is structured as follows. Section 2 discusses the alternative approaches of source-to-source rewriting, link-time rewriting and post-link-time rewriting. Section 3 presents the protections we implemented in a prototype, and discusses how to measure the overhead resulting from the link-time implementation rather than from the protections themselves. This overhead is evaluated in Section 4. Section 5 gives an overview of related work. Finally, Section 6 draws conclusions.

2 Compatibility with Third-Party Compilers

This section discusses the potential of source-to-source, link-time and post-link-time code rewriting in the scenario where security experts have to protect software that is to be compiled with third-party compilers. More specifically, the goal of the security experts is to transform the code such that it implements sufficient protection at minimal overhead.

2.1 Source-to-source rewriting

Source-to-source rewriters, be it tools or programmers, insert redundancy before the code is compiled. They do so by duplicating source code statements. In our eyes, source-to-source rewriters suffer from a number of major drawbacks in the targeted scenario.

First, optimizing compilers risk undoing the protection by eliminating the inserted redundancy, e.g., by means of common subexpression elimination [38]. To avoid this, i.e., to ensure that *sufficient* redundancy survives the compiler and remains present in the generated code, the source-to-source rewriter can either insert code that is not analyzed and optimized by the compiler, as is typically the case with inline assembly code, or he can insert code that is complex enough to withstand being identified as redundant by the compiler's analyses. For ensuring sufficient redundancy, both options work fine.

It remains a problem, however, how to ensure the presence of only the *necessary* redundancy, i.e., sufficient redundancy with minimal overhead. Based on our experience with several open source and proprietary compilers, we do not consider this feasible. Fundamentally, it is very hard, and most often simply not possible, to insert redundant code in the source that is not eliminated by the compiler, but that at the same time does not limit the precision or scope of the compiler's existing analyses and optimizations to reduce the code size or the execution time of the remaining code. Even if a kind of code complexity sweet spot existed where both properties hold, this sweet spot would be very dependent on the specific compiler and implemented protection. As a result, neither the protection nor the minimal overhead introduced by the source code rewriter would be portable. This implies that the source code rewriter, in case it is an automated tool, would have to be retuned, retrained or ported to each different combination of compilers and target architecture used. In case the rewriting happens manually, it means the manual effort would have to be reinvested every time a new compiler or target architecture is used. Moreover, it would also require the security expert to be a compiler expert. Certainly, that is not ideal.

Fundamentally, the sketched problem is an instance of a more generic problem of source-to-source rewriters: as security concerns many abstraction layers, incl.

physics, computer architecture, binary code and protocols, many security policies involve lower-level aspects that are hard to control in source code when the used compilers are black boxes. In the above instance, the problem is to specify which redundancy should survive the compiler and which complexity should not.

Furthermore, as source code rewriters are language-dependent, they need to be redeveloped for every language. Finally, white-box and black-box security testing typically takes place on the final binary code. Even though techniques have been developed to bridge the gap, communication between testing teams and protection tool developers is harder if the former are studying assembly code generated by one or more black-box compilers, while the latter are working on source code.

2.2 Link-time Binary Rewriting

Binary code rewriters or assembly rewriters that are applied to compiled or manually written assembly code do not suffer from the above problems. They do suffer, however, from the fact that they have to operate on code that lacks high-level, abstract semantic information, such as symbol and type information. This lack of information limits the precision and scope of many program analyses and transformations, and can hence have a negative impact on the provided level of protection as well as the incurred overhead.

From a security perspective, we first have to consider whether it is possible, with link-time rewriting tools, to reliably obtain *sufficient* protection on code generated with third-party compilers.

We conjecture that this is indeed possible. Regarding the wide range of transformations that can be applied at link-time, as necessary to insert the necessary redundancy, we point to the existing applications of the open-source link-time binary rewriting framework Diablo[1] [48]. These applications include optimization and compaction [19], obfuscation [4,34] and deobfuscation [32], anti-tampering [51], formal verification of binary code [54], instrumentation [16], GUI binary code editing [49], and operating system customization [12].

Diablo's potential for reliably transforming code generated with third-party tool flows and targeting different processor architectures is obvious given that Diablo has been applied to software written in different languages, incl. C, C++, assembly and Fortran; binary code generated with different compiler generations covering more than a decade of proprietary as well as open-source compilers (incl. ARM ADS, ARM RVCT, and gcc) [19]; system libraries that contain significant amounts of manually-written assembly, incl. newlibc[2], glibc[3], and uClibc[4] [19]; a range of architectures, incl. SH [14], PowerPC [10], MIPS [33], IA64 [5], x86 [12], and ARM incl. Thumb [12,19]; the Linux kernel [12, 11], which features artifacts such as mixed code for physical and virtual address spaces, privileged instructions, manually written assembly not adhering to the conventions as specified in application binary interfaces (ABIs), and a complicated, non-standard build process.

Despite these existing demonstrations of Diablo's flexibility and reliability, it is not clear *a priori* that a tool like Diablo can deliver acceptable protection against fault-injection at acceptable overhead, with an acceptable engineering effort. The reason is that all tools developed on top of Diablo have been designed to depend solely on information that is generally available in object files, such as symbol information, relocation information, and optionally debug information [30]. This enables them to handle code generated by open-source compilers as well as closed compilers, but it also limits the capabilities of link-time rewriters.

First, the link-time rewriters lack high-level semantic information about the code to be rewritten. For example, no type information is available, which makes alias analysis much less precise [20,38]. Consequently, link-time rewriters typically need to handle memory as a black box. For example, after reading in a program's object files that are normally linked with a standard linker, Diablo disassembles the code and builds a whole-program, interprocedural control flow graph (CFG) [38]. The nodes in this graph are basic blocks, i.e., single-entry single-exit sequences of instruction, in which instructions are represented using a very low-level, assembly-like intermediate representation (IR) [15, 38]. These instructions operate on type-less architectural registers that cannot be aliased because pointers to registers to not exist. With the exception of memory accesses at constant addresses and direct stack accesses, all loads and stores are handled as if they access one big memory array at unknown locations.

This is a big contrast with compilers, which perform part of their analyses and optimizations on code that operates on typed variables and objects. Only after the compiler has performed the high-level optimizations, the variables are assigned to registers or, when not enough registers are available at some point, they are spilled to memory. This register allocation can be optimized globally in a compiler [38].

Link-time rewriters do not have this optimization potential. When they rewrite and duplicate code, they

[1] `http://diablo.elis.ugent.be`

[2] `http://sourceware.org/newlib/`

[3] `http://www.gnu.org/software/libc/`

[4] `http://www.uclibc.org/`

will have to find free registers to store temporary values. In case they cannot find them, because the compiler has used all available registers for the original code, the link-time rewriter has to free registers by temporarily spilling values to the stack. Applying this spilling locally on a low-level IR with register operands potentially introduces considerably more overhead than what can be achieved in a compiler.

Secondly, at link time indirect control flow transfers, of which binary code analysis cannot resolve the targets precisely, have to be modeled conservatively (i.e., over-approximated) on the basis of relocation information [15,17]. Through additional edges in the CFG that model that over-approximation, the CFG models a superset of all possible executions of a program. This is safe, but leads to a loss in analysis precision for interprocedural analyses like context-sensitive liveness analysis, conditional constant propagation, copy propagation, and reachability analysis, on the basis of which tools like Diablo apply optimizations such as unreachable and dead code elimination, constant folding, inlining, etc. [38]. For a more detailed discussion, we refer to the existing literature [15,17,19,48].

Over the last decade, several techniques and tools have been proposed to improve the precision and scope of automated analyses of binary code, e.g., to retrieve targets of indirect control flow transfers [26] and to recover type information [21]. Moreover, advanced tools such as TSL (Transformer Specification Language) for automatically generating analyses based on abstract interpretation facilitate the engineering of new analyses [31]. Still, because of (1) the wide range of transformations and optimizations that compilers apply, including in some cases obfuscating transformations, because of (2) the undecidability of many compiler analyses, and because of (3) the limitations one needs to impose on the resources and time available to analyze programs, there will necessarily always be limitations on the precision and scope of automated analyses.

It is mainly because of these limitations that it is necessary to evaluate the extra overhead introduced by applying hardening transformations at link time, i.e., the overhead that could be avoided by instead applying the same transformations in source code, in a compiler, or by manually rewriting assembly code.

2.3 Post Link-time Binary Rewriting

O'Sullivan *et al.* recently proposed an interesting alternative for link-time rewriting [40]. Their SecondWrite tool rewrites a binary executable lacking relocation information post link time. The executable is first disassembled into the IR (intermediate representation) of LLVM[5] [29], then rewritten and optimized, before being scheduled and register-allocated into a new binary. This offers many of the benefits of link-time rewriting, and in addition end users can rewrite any untrusted third-party software binaries, even when the binaries lack any kind of symbol or debug information. Furthermore, working with a more abstract compiler IR allows SecondWrite to alter stack frame layouts globally and to apply all existing optimizations in LLVM for free. At least in theory, SecondWrite hence supports more elaborate protection schemes. Furthermore, a more global register allocation can be performed on the LLVM IR, which potentially reduces the overhead of applied fault-injection protections.

However, SecondWrite's approach also comes with some major disadvantages. Its main disadvantages relate to the need to maintain conservativeness in the absence of relocation information [45]. To ensure that operations involving computed addresses (e.g., indirect branches, function pointers and memory accesses through pointers) are rewritten correctly, SecondWrite does not relocate any statically allocated data. The read-only data embedded in the original code section (e.g., address or constant pools) are not relocated either, and remain located at their original addresses. Furthermore, before indirect control flow transfers, SecondWrite inserts translation table lookups to translate addresses from the original binary to addresses in the new binary at run time, such that the transfers are redirected correctly even if their targets could not be resolved statically.[6] In addition, whenever SecondWrite cannot prove that some bytes embedded in the code section are actually data, it conservatively handles these bytes as both code and as data. To that extent, SecondWrite leaves a copy of the bytes in their original location to support all potential data accesses to the bytes, and adds a rewritten copy of the bytes interpreted as code to the program IR. So in the address space of the rewritten binary, the statically allocated data sections remain in place, as do the (potential) data pools from the original code section. Some data is duplicated as code and in addition translation tables and lookups are added. This all results in considerable code size and file size overhead, even before any protection is applied.

Other post link time rewriters like REINS [50] and the tool by Zhang and Sekar [53] suffer from similar drawbacks as SecondWrite, in that their handling of indirect control flow and memory accesses also inflates

[5] `http://llvm.org/`

[6] Note that this feature has nothing to do with enforcing control flow integrity. It only relates to ensuring that the application behaves the same before and after rewriting, whether that is as intended by the developer or not.

the executable size even before any actual code hardening has been applied.

In many scenarios, that inflation is not problematic. All post-link-time rewriter developers target and evaluate desktop computers and applications [40,45,3, 53,50], where code size and file size are not really concerns. Furthermore, with desktop operating systems, all rewritten code can simply be reallocated contiguously in a completely different range of the address space.

Smart cards, however, typically feature very small memories and often lack virtual memory. As a result, smart card software developers cannot rely on any of the existing post-link-time rewriter implementations.

That does not imply, however, that post-link-time rewriting approaches are fundamentally infeasible. By including more advanced binary code analysis steps like the ones referenced near the end of Section 2.2, and by ensuring that only relatively standard, clean binary code is generated for an application, a post-link-time rewriter should be able to correctly relocate all data and code, thus avoiding the overhead of the current approach of SecondWrite and the other tools. The generation of such clean binary code, i.e., code with recognizable control flow transfer patterns, can most often be realized by imposing coding guidelines and by controlling the compiler options. In some cases however, it might be more tricky, e.g. where it concerns system libraries that mix C code and inline assembly fragments.

As the same reasoning can be applied to link-time rewriters, the post-link-time approaches are ultimately as hampered as link-time solutions with regards to their potentially imprecise modeling of indirect control flow.

So fundamentally, the main difference between tools like SecondWrite on the one hand and tools like Diablo on the other hand, is the abstraction level at which code is represented and transformed. Regarding the hardening of code against fault-injections, it is an open question whether operating at a compiler IR level instead of at an assembly IR level offers real benefits.

3 Implemented Sample Protection Schemes

For this study, we implemented four first-line-of-defense cookbook protections that provide local hardening against single-instruction failure attacks [37]. To understand the value of these schemes and the rationale behind some implementation choices and practical issues, it is important to consider the attack model in which these protections are considered.

3.1 Attack Model

By means of offline hardware and software hacking in labs, attackers reverse-engineer the complete software installed on a smart card, incl. its (often deterministic) timing. This enables them to engineer fault-injection attacks to steal secret data, e.g., using hacked terminals at restaurants or ATMs, and to counterfeit credit cards. To deploy the attacks covertly, the fault-injection infrastructure and the data acquisition hardware need to fit into small devices that are, with respect to outside appearance and observable timing behavior, not distinguishable from standard terminals by a layman.

An important form of fault-injection attacks in such scenarios are single-cycle faults. Knowing the software, the attacker selects an instruction to disrupt. Knowing the software timing, he then determines the cycle in which to inject a fault, e.g., by temporarily lowering the supply voltage or by flashing the instruction fetch bus with a laser. Which exact bits will flip because of the injected fault is typically not under control of the attacker, however.

By causing random bit flips on a chosen branch instruction, an attacker has a good chance of turning it into an operation that does not transfer control, but instead falls through to the next instruction. This follows from the fact that most instructions (i.e., most instruction encoding bit combinations) fall through. Similarly, an injected fault can turn a memory store into a non-store with high probability. And an instruction that sets a condition flag can likely be converted into one that does not set the flag. By contrast, with random bit flips an attacker has no real hope of turning an arithmetic logic unit (ALU) instruction into a control flow transfer to a chosen address, or into a store operation at a chosen memory location. This severely limits the likely successful attacks he can mount.

Besides failing to perform its intended task, a faulted instruction will of course still have some unpredictable effect on the processor state, such as overwriting a register value. But each possible effect has a relatively low probability. Furthermore, there is a relatively low chance that the effect of the faulted instruction will completely corrupt the continuing execution of the program. For example, if a branch instruction is faulted, it is not unlikely that the only result will be an overwritten register whose value is no longer used in the program. From the attacker's perspective, a faulted control flow transfer, memory operation or condition flag setting operation will hence quite likely behave like a simple no-op instruction.

In summary, we envision attackers will try to identify a branch, call, return, compare, or store instruction

that, when turned into a no-op, causes sensitive data to be leaked or unauthorized tasks to be performed. The first goal of the hardening protections is therefore to rewrite the code such that replacing single instructions by a no-op no longer causes unwanted behavior.

The existing protections for which we implemented support in our prototype rewriter achieve this for certain classes of code fragments. We certainly do not want to claim that we present new protection schemes, or that our prototype can offer full protection. We only implemented a number of basic protections to demonstrate that the automation of these types of protections in a link-time rewriter is feasible and comes with an acceptable overhead.

Nonetheless, we want to argue that even the application of only the implemented protections can be useful in certain scenarios. In particular, the implemented protections can contribute in lowering the likelihood of a successful attack to the point where an attacker's costs (e.g., the hacked terminals, the offline analysis investigation, bribing people to cooperate, ...) outweigh his likely benefits. That is the reason why rather simple forms of redundancy are a typical first line of defense in the first place [37].

More recently, multiple fault injection attacks have also become viable [47]. The protections presented in here do not necessarily protect against those.

3.2 Implemented Protections

We implemented redundancy-based protection schemes in a tool on top of the Diablo framework. We targeted the ARM Cortex-M0 instruction set architecture (ISA) of the ARM SecurCore SC000 processors [52], which are aimed at future smart card applications. In these schemes, the protected program executes some *invalid state exception* code when a fault is detected. In our prototype, we opted for implementing that invalid state exception code by means of simple infinite loops, which at least prevent that any secret data is outputted following a fault injection. Given Diablo's existing support to inject user-provided code into a program [16], this loop can easily be replaced by other invalid state handling code that is, e.g., selected not to leak information via timing side channels, but otherwise continue the normal execution of the program.

We again want to emphasize that the implemented protections are not meant to demonstrate full protection or stronger protection than what can be achieved manually today. They were instead chosen because they (1) are being used in practice, (2) allow us to pinpoint technical and usability issues, and (3) allow us to evaluate the feasibility and overhead of automated, link-time hardening transformations compared to manual or compiler-based hardening.

```
        ldr r0,[r1]
        cmp r0,#5
        beq .Lsafe
.Lsens:
        <sensitive code>
.Lsafe:
        <safe code>
```

(a) Original code

```
        ldr r0,[r1]
        cmp r0,#5
        beq .Lsafe
        ldr r0,[r1] // duplicated
        cmp r0,#5   // duplicated
        bne .Lsens  // duplicated
.Lfault:
        <invalid state exception>
.Lsens:
        <sensitive code>
.Lsafe:
        <safe code>
```

(b) Protected code

Fig. 1 Example of conditional branch duplication

3.2.1 Conditional Branch Duplication

Sensitive code paths are often shielded by checks that verify whether a correct password was entered, the correct key was used, a valid request was performed, etc. On smart card processors like the ARM SecurCore SC000, these checks correspond to conditional branches in the binary code. The branches are taken or not taken depending on flags in a flags register, which can either be set explicitly by a comparison instruction or implicitly according to the result of an ALU operation. Attacks can focus on the input values used to perform the operation that sets the flags, on that operation itself, or on the conditional branch that depends on the flag.

To protect against attacks that make the checks ineffective by skipping one of those instructions, we duplicate the computation of the flags and the conditional branch. A typical scenario targeted by this transformation is depicted in Figure 1(a)[7]. In this case, we want to prevent that an attacker can force the conditional branch to fall through by either skipping the branch itself, or by skipping one of its immediately preceding instructions. In the original code, and in a fault-free operation, the conditional branch `beq` falls through to the sensitive code if and only if the value loaded from address `r1` into register `r0` by the `ldr` instruction is not

[7] A brief overview of ARM & Thumb architecture is given in the electronic appendix to this article

```
        ldr r0,[r0]
        cmp r0,#5
        beq .Lsafe
.Lsens:
        <sensitive code>
.Lsafe:
        <safe code>
```

(a) Original code

```
        ldr r2,[r0]
        cmp r2,#5
        beq .Lsafe
        ldr r0,[r0] // duplicated
        cmp r0,#5   // duplicated
        bne .Lsens  // duplicated
.Lfault:
        <invalid state exception>
.Lsens:
        <sensitive code>
.Lsafe:
        <safe code>
```

(b) Protected code when register `r2` is free

```
        push r2     // potentially avoidable spill
        ldr r2,[r0]
        cmp r2,#5
        pop r2      // potentially avoidable spill
        beq .Lsafe
        ldr r0,[r0] // duplicated
        cmp r0,#5   // duplicated
        bne .Lsens  // duplicated
.Lfault:
        <invalid state exception>
.Lsens:
        <sensitive code>
.Lsafe:
        <safe code>
```

(c) Protected code when no free register is available

Fig. 2 Conditional branch duplication that needs additional temporary registers or stack space.

```
        lsl r0,r1
        cmp r0,#5
        beq .Lsafe
.Lsens:
        <sensitive code>
.Lsafe:
        <safe code>
```

(a) Original code

```
        mov r2,r0  // unavoidable data copying
        lsl r0,r1
        cmp r0,#5
        beq .Lsafe
        lsl r2,r1  // duplicated
        cmp r2,#5  // duplicated
        bne .Lsens // duplicated
.Lfault:
        <invalid state exception>
.Lsens:
        <sensitive code>
.Lsafe:
        <safe code>
```

(b) Protected code when register `r2` is free

```
        push r0    // unavoidable data copying (stack)
        lsl r0,r1
        cmp r0,#5
        pop r0     // potentially avoidable spill
        beq .Lsafe
        lsl r0,r1  // duplicated
        cmp r0,#5  // duplicated
        bne .Lsens // duplicated
.Lfault:
        <invalid state exception>
.Lsens:
        <sensitive code>
.Lsafe:
        <safe code>
```

(c) Protected code when no free register is available, but where `r0` is dead on entry to the safe code.

Fig. 3 Another conditional branch duplication that needs additional temporary registers or stack space

equal to 5. If the load operation (`ldr`) is skipped by a fault injection, however, the compare will be based on an older value of register `r0`. In that case, the conditional branch may fall through to the sensitive code even when the value at address `r1` in memory was 5, thus potentially leaking sensitive data. In case the compare operation `cmp` is skipped by a fault injection, the conditional branch will fall through or jump based on the condition flags already set before the `ldr` instruction. So again, the branch might fall through even when the value at address `r1` in memory was 5.

The code is therefore transformed into the code of Figure 1(b). In this code, skipping a single instruction will never allow the attacker to reach the sensitive code when the value at location `r1` in memory is equal to 5: If any of the first three instructions are skipped, the duplicated load and compare instructions will effectively make the branch-if-not-equal instruction `bne` fall through into the invalid state exception code. By skipping one of the duplicated instructions, the only thing the attacker can achieve is that the conditional branch falls through instead of being taken. So again, the attacker cannot force the sensitive code to be reached.

More complex variations of the code in Figure 1(a) can occur. First, there can be multiple definitions of the input value(s) of the comparison in different predecessors of the comparison's basic block. The duplication of the definitions can then be skipped, which weakens the protection because of reduced redundancy. Alternatively, the multiple definitions can be duplicated, possibly by means of tail duplication [38], even before the actual redundancy is being inserted. So far, our prototype only supports the former.

Secondly, it does not always suffice to merely duplicate instructions. Consider the case when an instruction's destination operand register is also one of its source operands, as in the `ldr r0, [r0]` instruction in Figure 2(a). The problem is that the instruction overwrites the value of its own source operand, which is hence no longer available for the duplicated instruction. In such cases, we need to find one or more free (i.e., dead) registers to rename registers before performing the duplication, or to store copies of the original source operand values temporarily. Our prototype finds such free registers by means of a state-of-the-art bidirectional context-sensitive interprocedural liveness analysis [18]. When free registers are found, we can simply use them, as shown in Figure 2(b). When no free registers are found, they are created by either renaming registers locally, or by inserting the necessary spill code instructions that temporarily save values on the stack, as with the `push` and `pop` instructions in Figure 2(c).

Please note that in this example, register `r2` can be used directly because in the Thumb ISA, `ldr`'s destination operand is not also an implicit source operand. For other instructions, however, including most ALU operations, the destination operand is also an implicit source operand. This is because Thumb's instruction width of 16 bits is too narrow to encode three different register operands in all instructions. When such instructions with an implicit source operand need to be duplicated, an additional copy operation needs to be inserted to duplicate the operand's original value before it is overwritten. Figure 3(b) shows an example of such an additional `mov` instruction, which cannot be avoided. Sometimes, depending on the liveness of the involved registers, this copy operation can be fused into the spilling code. An example is shown in Figure 3(c), where the value in `r0` is copied to the stack by the `push` instruction.

Apart from spilling registers to free them, it can also be necessary to spill the flag register when the flags are consumed by instructions computing the flag for the conditional branch. Spilling the flag register itself by means of `msr` and `mrs` instructions adds additional overhead, as might the freeing of normal registers to spill the flag register to. This is illustrated in Figure 4. This code example also illustrates that setting condition flags in Thumb is not limited to simple compare instructions. Instead, almost all Thumb instructions (sometimes optionally) set the flags. This further complicates finding the transformation with the lowest overhead.

On the ARM Cortex-M0's with its Thumb ISA, sometimes a form of "light" spilling can be used to avoid the power-consuming memory operations of traditional spilling. In this ISA, only the lower registers

```
    sbcs r3,r0    ; subtract with carry,
                    set Z flag if result is zero
    beq  .Lsuccess
    <failure handling code>
.Lsuccess:
    <sensitive code>
```

(a) Original code

```
    push  r3,r4    ; free r3 and r4
    mrs   r4,cpsr  ; temporarily save carry flag in r4
    sbcs  r3,r0
    beq   .Ldup1
    add   sp, #8   ; abandon saved r3 and r4
    <failure handling code>
.Ldup1:
    msr   cpsr, r4 ; restore saved carry flag
    pop   r4,r3    ; restores r3 and r4
    sbcs  r3,r0
    beq   .Lsuccess
    <invalid state exception>
.Lsuccess:
    <sensitive code>
```

(b) Protected code

Fig. 4 Conditional branch duplication with additional overhead for duplicating condition flags

`r0`–`r7` are generally accessible as source and destination operands in all instructions. Registers `r8`–`r12` are accessible through fewer instructions, incl. moves and compare instructions without immediate operands. When the registers `r8`–`r12` or `r14` (which holds the return address) are dead at some program point, they can serve as spill locations for lower registers that need to be freed to duplicate operations.

While the above examples are by themselves pretty straightforward, they illustrate that finding the code with the least overhead involves many conditions to be checked. It is hence clear that manual application of the protections, e.g., on assembly code is cumbersome and time consuming, even if only local spilling options are considered. So on the one hand, automation, e.g., through binary code rewriting, is highly welcome. On the other hand, it is also clear that when the automatically computed liveness information lacks precision to find the needed free registers, significant additional overhead might be introduced.

In Section 4, we will estimate that additional overhead. For that estimation, we will count the following instructions as *unavoidable overhead* of the hardening of conditional branches:

- the duplicated instructions that implement the hardening;
- `mov` instructions inserted as in Figure 3(b) to copy source values that are unavoidably overwritten by an operation to be duplicated because that opera-

tion's destination operand is also an implicit source operand;
- `push` instructions that are fused with such `mov` instructions, and that hence implement not only spilling, but also the necessary copying;
- `msr` and `mrs` operations needed to duplicate condition flags because an instruction that both consumes and overwrites them is to be duplicated as part of the hardening.

These encompass all instructions that are unavoidable given the limitations of the Thumb ISA and the requested code duplication to implement the hardening.

By contrast, we will count all other inserted spill operations and all other move operations as *potentially avoidable overhead*. The reasoning is that some of these spills and moves might have been avoided with more accurate liveness information or through more global register reallocation or more advanced transformations than are currently supported by our tool. In other words, these spills and moves might have been avoided during manual hardening of assembly code or when the hardening was done by a compiler in the yet-to-be-allocated compiler IR of the code.

Our estimate of the potentially avoidable overhead will certainly overestimate the *truly avoidable overhead*.[8] As such, our bookkeeping allows us to put an upper bound on the extra overhead potentially caused by the link-time rewriter's immaturity or by its lack of analysis precision and global register allocation. In Section 4, this will ultimately allow us to demonstrate the feasibility of link-time smart card code hardening.

Our tool, being just a prototype, only targets the most easily attacked types of code fragments, i.e., the low-hanging fruit of attackers. Most importantly, our implementation currently assumes that the fall-through path following a conditional branch is the sensitive one. This follows from the insight that it is much simpler for an attacker to skip a branch that had to be taken under fault-free conditions, than it is to force taking a branch that would not be taken under fault-free conditions. If sensitive code lies on the branch-taken path, the branch has to be inverted first, or an additional inverted check has to be inserted on the fall-through path as well. Our current prototype does not yet support this. In Section 3.3, we discuss how a programmer can inform a link-time tool about the exact program points to be considered sensitive and on the policies that should be applied at each of them.

[8] The truly avoidable overhead can only be determined precisely by developing a hardening compiler or by applying all protections manually, for which we lack the time and resources.

Furthermore, our prototype branch duplication skips conditional branches at loop exit points [38], because loops are often better protected with other transformations such as the one discussed in Section 3.2.4.

In addition, our duplication of instructions is currently limited to the following two instruction patterns, for which we duplicate 3 and 2 instruction respectively:

pattern 1:	1)	any instruction producing `rX`
	2)	`cmp`/`tst` `rX, #immediate`
	3)	conditional branch
pattern 2:	1)	any instruction setting the flags with operands `rX, rY` or `rX, #immediate`
	2)	conditional branch

We opted for these patterns because they make up over 95% of all conditional branches in our tested benchmarks and because they cover both simple and complex scenarios as discussed above. Implementing support for more patterns and for duplicating more instructions would involve a significant effort without significantly changing the outcome of our experiments in Section 4.

3.2.2 Call Graph Integrity

Security analyses performed on a program's call graph are only as trustworthy as the guarantee that only modeled calls or returns can occur. By injecting bogus call or return addresses into the execution of a program, it is possible to invalidate any call graph constructed statically.

The call graph integrity transformation we implemented works at a local level: at each individual call and return site a value is set that can be checked at the intended destination. At the start of every function and at every return site, it is then possible to verify that control indeed came from one of the allowed source locations. This in effect prevents calls and returns from being skipped. This is less strong than existing protections that verify entire call chains, but on the plus side it can be easily applied to call graphs that contain hard to analyze constructs such as recursion.

Since our transformation is applied at link time, supporting indirect function calls through function pointers or polymorphic method invocations requires extra care. Lacking type information, link-time rewriters typically cannot determine the exact targets of an indirect call. This is solved by clustering all functions that can be called indirectly according to the symbol and relocation information in the object files and treating them as a single function as far as this transformation is con-

```
Caller:
     ...
     mov r4,#eb
     blx CalledFunction
     ...
Callee:
     cmp r4,#eb
     beq .Lsuccess
     <invalid state exception>
     ...
.Lsuccess:
     <sensitive code>
```

Fig. 5 Passing and checking an ID

```
     str r0,[r1]
     ldr r2,[r1]
     cmp r0, r2
     beq .Lok
     <invalid state exception>
.Lok:
     ...
```

```
     strb r0,[r1]
     ldrb r2,[r1]
     eors r2,r0
     lsls r2,#24 // sets or resets flag
     beq .Lok
     <invalid state exception>
.Lok:
     ...
```

Fig. 6 Two examples of a store followed by its verification

cerned. While this makes the protection less tight, it allows us to deal statically with uncertainties introduced by dynamic program behavior.

Figure 5 shows how each check consists of two parts. Before every call a register or global variable is set to a unique call identifier of the callee or cluster of callees. In the example, the identifier has the value 0xeb. Next, instructions inserted in each function's prologue verify whether the set value matches its identifier. Similarly, before every return instruction a register or global variable is set to a different return identifier. This value is checked at the return points in the callers.

The hardening process starts by partitioning the program's functions into clusters whose members can call each other indirectly, that can be called from the same indirect call site, or that are linked through interprocedural gotos [15,17].

Next, the registers free on entry and exit in all functions in a cluster are collected. If some register is always free on entry, it will be used to pass the value from the caller to the callee; otherwise the value is passed via a global variable. A similar decision is made for the exit, independently of what was decided at the entry. The registers used to pass identifiers can obviously differ across clusters.

Diablo also records whether the condition flags are free on entry/exit of a cluster's functions. Their availability is important because the checks involve comparison instructions that overwrite them. While no ARM ABI guarantees that the flags maintain their value across function boundaries, functions written in assembly do not always adhere to conventions. So whenever they are not provably dead at some call or return point, they are temporarily stored in a free register, which is created through register spilling when necessary. In a large set of programs we examined, only the ABI-defined functions for emulating floating-point comparisons on systems lacking floating-point hardware return their result via the condition flags.

For evaluating the efficiency with which this hardening technique can be applied automatically at link time, we consider all `mov` instructions that set the identifier values as well as all compare and conditional branch instructions that check that value and jump to infinite loops, as unavoidable. All other inserted instructions, such as spills and `msr` and `mrs` instruction, are counted as potentially avoidable overhead.

3.2.3 Memory Store Verification

The failure of a store operation at run time generally means that program state is lost. This can be addressed by checking that the correct value was written to memory. Such a check also introduces some resiliency to memory errors.[9] The proper execution of a store can be verified by loading the stored value back from memory and by comparing it to the value that should have been stored, as depicted in Figure 6. This transformation requires an extra free register (`r2` in the example) to reload the stored value. Spilling might be needed to create such a register. When a 16-bit halfword or a byte are being stored as in the example at the bottom of Figure 6, some additional instructions are needed because the reloaded halfword or byte cannot be compared directly against the 32-bit word out of which the halfword or byte was stored.

Additionally, we have to ensure that the inserted comparison does not overwrite any condition flags live after the store to be protected. To keep all live flags' values, we can simply save all of them temporarily, as was done for the conditional branch duplication in Figure 4. Alternatively, we can often insert an instruction that recomputes the live flags after the store instruction, thus freeing the flag at the point of the store. Fig-

[9] The presented verification technique aims to ensure that write operations take place as intended. For protecting the values once they are stored in memory, complementary techniques such as error-correction codes can be used [23].

```
        sub r0, #1 // sets the condition flag
        str r0, [r1]
        beq .Llabel // based on subtraction result
```

(a) Original code

```
        sub r0,#1 // sets the condition flag
        str r0, [r1]
        ldr r2, [r1]
        cmp r0, r2
        beq .Lcorrect
        <invalid state exception>
.Lcorrect
        cmp r0, #0 // recomputes the Zero flag
        beq .Llabel
```

(b) Protected code

Fig. 7 Example of memory store verification with flag recomputation

ure 7 depicts an example, in which the combination of `msr` and `mrs` (and the potentially necessary spilling to find a register for them to use) are avoided by inserting a simple compare operation that recomputes the Zero flag. Furthermore, it can also be useful to reschedule the existing code, such that more free registers become available right after the store to be protected.

For evaluating the avoidable overhead of this hardening technique applied automatically at link time, we count the load, the comparison (or alternatively the exclusive or and the shift), and the conditional branch as unavoidable. All other operations are considered potentially avoidable, including spills and `msr` and `mrs` instructions or their alternative comparison to recompute flags. We count `msr`, `mrs` or their replacement as potentially avoidable because the store instructions to be protected can often be moved to points in the schedule where no condition flags are live, and hence where no flag saving and restoring instructions are needed at all. So if our prototype tool does insert those instructions, it might be doing so unnecessarily, because it lacks more aggressive code motion support or because it lacks precise enough liveness information.

For hardening instructions that store to the stack outside of function prologues and epilogues, we will count the inserted spill instructions as *potentially, but unlikely avoidable.* The reason is that most stores to the stack in the original program result from register pressure. So for most of the stores to the stack, the presence of the store itself indicates that even in the original, unhardened code, the compiler could not find enough free registers. So for most of such stores to the stack, spill code inserted to free a register will in fact be unavoidable. A tiny fraction of the stores to the stack are present, however, independently of register pressure, i.e, because their stored values escape their stack frame or because it concerns accesses to `structs`. As we have no method to differentiate 100% accurately between the two cases, we cannot accurately count the number of unavoidable spills inserted to protect stores to the stack. But we can estimate them, and hence we report them as potentially, but unlikely avoidable.

3.2.4 Loop Iteration Counter Duplication

Many applications are sensitive to the number of iterations executed in their loops. For example, running too many iterations of a loop exporting a buffer might result in sensitive data stored after the buffer being exported, and running too few rounds of an encryption algorithm might leak information about a secret key. Therefore we need to protect loops such that they execute precisely the foreseen number of iterations. We implemented a loop iteration counter duplication to achieve this. Its protection on the simple loop of Figure 8(a) is depicted in Figure 8(b).

For this protection, the initial loop counter value (`r1` in Figure 8) is duplicated (into `r5`) in the loop's preheader. In the loop body, the loop counter operation (`sub`) needs to be duplicated. Moreover, both directions following the conditional branch at the exit of the loop need to be protected to ensure that not too few and not too many iterations are executed. That is why two checks have been inserted in the code of Figure 8(b).

In our current prototype implementation, we protect natural loops of which the fault-free number of iterations can be determined before the loop is entered. This requires that the loop contains a counter that is updated exactly once per iteration and that the loop exit is controlled by comparing the loop counter to a loop invariant value. This excludes, e.g., loops that iterate over dynamic data structures such as linked lists. For such loops, more generic loop control protection can be implemented by means of the already presented protections. Furthermore, our prototype does not yet support function calls in loop bodies.

For this loop hardening technique, it is more complex to estimate which of the inserted instructions are unavoidable, and which ones might constitute avoidable overhead because they stem from a lack of precise liveness analysis information and from our tool's inability to perform global register allocation. The reason is that in this case, the liveness ranges corresponding to the duplicated instructions span whole loop bodies, which are much larger code regions than those involved in the other hardening techniques.

For this protection we hence first estimate whether or not a loop's original code already suffers from register pressure. The idea is that when the original loop body

```
.Lpreheader:
     mov r2, r0
.Lbody:
     lsl r2, #1
     sub r1, #1
     cmp r1, #0
     bg .Lbody
.Lafter:
     ...
```

(a) Original loop code

```
.Lpreheader:
     mov r2, r0
     mov r5, r1
.Lbody:
     lsl r2, #1
     sub r1, #1
     sub r5, #1
     cmp r1, #0
     bg .Lcheck2
.Lcheck1:
     cmp r5, #0
     ble .Lafter
     <invalid state exception>
.Lcheck2:
     cmp r5, #0
     bg .Lbody
     <invalid state exception>
.Lafter:
      ...
```

(b) Protected loop code

Fig. 8 Example of loop iteration counter duplication

indicates that the original compiler or the assembly programmer were not able to allocate all used values to the best suited registers, we can assume that a security researcher protecting code manually will not be able to allocate the duplicated counter to the best suited registers either. In that case, it is impossible to avoid operations that store and load the duplicated counter to and from the stack before and after the duplicated update and compare instructions. Hence we should not consider such spilling operations a potential result of our tool's limitations.

For the original loop counter, which needs to be updated in the loop body, the best suited registers are `r0`–`r7`. In case the loop invariant value to which the loop counter is compared for exiting the loop, is not an immediate operand, the best suited registers to hold that invariant value are `r0`–`r12` and `r14`, because all of them can be accessed by compare instructions. When we detect that any loop invariant value of the loop or the original loop counter itself are not allocated to those best suited registers, we consider the original loop as being generated under register pressure. Any spill or "light" spill operations of the duplicated counter are then unavoidable. In case an original loop body does not show an unambiguous sign of register pressure, we consider all spill operations and "light" spills as potentially avoidable overhead.

This accounting method over-conservatively estimates that for any original loop without the above signs of register pressure, the rewritten loop (operating on the same values plus a duplicated counter) would not experience register pressure either. This is of course not always the case. This method therefore incorrectly counts some unavoidable spills as potentially avoidable. In other words, like the accounting method of the other transformations, this accounting method also overestimates the real overhead resulting from our tool's limitations.

3.3 Specifying Where to Apply Protections

Just as source-to-source tools suffer from the difficulty to specify security policies in terms of lower-level code properties, the previous sections might have given the impression that binary rewriting tools suffer from a similar problem in terms of higher-level source code constructs and statements. It is therefore important to note that the policy specification does not need to be limited to the binary code or assembly code abstraction levels.

With a more complete tool than our prototype, programmers would be able to annotate their source code to specify program points at which certain policies have to be applied. They would then be able, e.g., to specify which paths following conditional branches are sensitive (see Section 3.2.1). This can easily be supported by means of source code annotations in the form of pragmas and attributes similar to the ones already supported by many compilers today. Consider the function `f()` in Figure 9 that is annotated with the existing `noinline` attribute to tell the GCC compiler not to inline that function [38], and in which the conditional branch is annotated with the `unlikely` built-in macro that informs the GCC compiler to optimize the code under the assumption that the branch is likely not taken [38]. Very similar annotations like `__protect_taken` or `__protect_call_graph_integrity` could be inserted to inform a link-time rewriting tool about the code fragments and paths to be protected. Such annotations provide a relatively convenient way for the programmer to specify his security policy, without him having to insert any protection manually.

To make this work, a precompiler tool will extract and omit the annotations from the source code and store them for later use in terms of source line numbers. The existing compiler can compile the source code into binary code that is annotated with standard debug information that maps addresses in the binary code to

```
__attribute__((noinline)) void f(int x) {
   if (unlikely(x==0)) clean_buffer();
   else output_buffer();
}
```

Fig. 9 GCC source annotations

source code line numbers. On the basis of that information, the link-time rewriter would then apply the specified and extracted policies. At the time of writing, we have completed support for specifying policies for full functions or files, and for selecting types of fragments or paths to which to apply a policy. For example, the conditional branches preceding loop exit edges can be excluded from the conditional branch duplication of Section 3.2.1, because loops are typically better protected with specific optimizations such as the loop iteration counter protection of Section 3.2.4.

4 Experimental Evaluation

Because of confidentiality, real smart card software is not publicly available. So instead we evaluated the coverage and overhead of our protections on C benchmarks from the embedded MiBench suite [22]. We compiled and protected ten MiBench benchmarks (see Table 1) for a semi-hosted simulation environment (QEMU 1.0 [9]). We used them to verify correctness and to collect execution profiles in order to measure the dynamic protection overhead, i.e., the overhead in terms of number of executed instructions. In the results, we will refer to these benchmark versions as s1–s10.

Whereas the crypto benchmarks s6–s8 in our benchmark suite implement functionality typical for smart card software, the other benchmarks were chosen (arbitrarily) to increase the sample set size of our experiments. Some of those benchmarks rely heavily on procedure pointers and floating-point emulation, and some benchmarks are IO-intensive. With respect to those aspects, they are not representative of typical smart card software. We still include them in our experiments because they increase the sample set size for evaluating the protections that are orthogonal to these aspects.

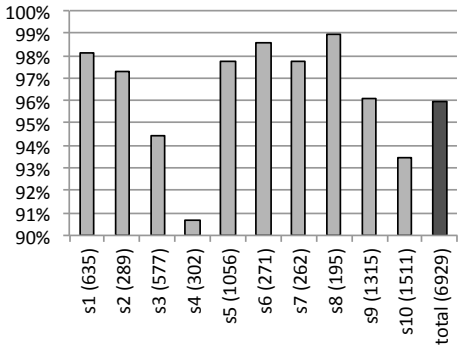
We used the ARM RVCT 4.1 compiler for ARM Cortex-M0 platforms at optimization level -O2. This compiler is a centerpiece of Keil[10], a development tool box that includes support for industrial smart card software development. We manually checked the generated code for the presence of the protections.

All binaries are linked statically, such that they include all needed RVCT 4.1 C library code. Via that library, our benchmarks contain considerable a-mounts of hand-written assembly code, which is not atypical for real smart card applications.

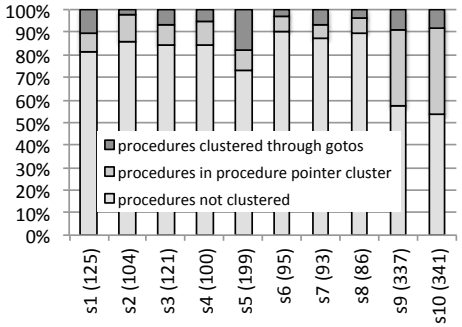
[10] http://www.keil.com/smartcards

Table 1 Benchmarks

benchmark	domain	short-hand	code size (bytes)	# ins
basicmath_small	floating-point	s1	19580	8266
bitcnts	integer bitcounting	s2	8204	3461
qsort_large	3D point sorting	s3	14824	6489
qsort_small	string sorting	s4	8012	3506
susan	image processing	s5	32172	14569
aes	crypto	s6	37092	12724
sha	crypto	s7	7296	3105
bf	crypto	s8	7792	2469
cjpeg	JPEG encoding	s9	54564	23088
djpeg	JPEG decoding	s10	61304	26673



(a) fraction of conditional branches duplicated



(b) clustering of procedures for call graph integrity

Fig. 10 Coverage results for conditional branch duplication and call graph integrity checking.

Whereas developers of real smart card applications would apply protections to only their sensitive parts, there is no notion of sensitivity in our benchmarks. They only serve the purpose of estimating the overhead of protections applied at link time, and of validating their correct application. Our tool hence applied the protections to the whole programs.

4.1 Coverage

4.1.1 Conditional Branch Duplication

Figure 10(a) shows the fraction of all conditional branches (excl. loop exits) that were duplicated with the trans-

formation described in Section 3.2.1. The absolute numbers of duplicated conditional branches are given on the X-axis. All coverage results are based on static counts. From all conditional branches excl. loop control, a large fraction of 96% is duplicated. Per benchmark, this varies between 90% and 99%. As for the small fraction of branches not duplicated, this was mainly the result of not finding the flag-setting instruction in the branch's basic block, as required by the instruction patterns targeted by our prototype.

From this we can conclude that even with our relatively immature prototype, the vast majority of the conditional branches can already be protected. Moreover, as we protected close to 7000 conditional branches in our experiments, our sample set is large enough to draw conclusions on expected overhead in Section 4.2.1.

4.1.2 Call Graph Integrity

For this protection, our prototype adds checks at all procedure entry points, and at all return points following call sites. So the coverage is 100%. In all cases, the cluster identifier is passed through a register. This is possible because in ARM Thumb, register `r8` is not generally accessible and not callee-saved, and hence it was not live at any procedure entry or exit point.

Figure 10(b) shows to what extent procedures are clustered as discussed in Section 3.2.2. The total number of procedures per benchmark is given on the X-axis. The vast majority of them (i.e., more than 80% for most benchmarks) are not clustered. Those procedures get unique identifiers, which provides strong protection.

In all benchmarks, we observed one non-singleton cluster containing all procedures that can potentially be called indirectly through procedure pointers[11]. Typically, this cluster contains 5–10 standard library procedures. In cjpeg (s9) and djpeg (s10), however, it contains more than 30% of all procedures, which moreover are not library code, but application code. For each image format supported by the benchmark programs, a structure is initialized with procedure pointers to routines that handle the supported image formats. This structure is then dereferenced throughout the (de)coding process, in what basically constitutes C-style polymorphic procedure calls. The resulting clustering significantly reduces the provided level of protection, as all of these functions now share the same two identifiers: one for their entries and one for their exits.

[11] In our link-time rewriter, the set of functions that can be invoked through function calls is conservatively approximated by computing the set of functions whose absolute address is stored or can be computed somewhere in the program, as indicated by the available relocation information.

Several observations have to be made here. First, we consider this result not problematic for our case, as we deem such heavy use of polymorphic procedure calls not representative for real smart card applications. Secondly, and more importantly, even if this form of call graph integrity would be applied and optimized manually, clusters would have to be formed when polymorphic procedure calls are present, albeit smaller clusters than the ones now considered by Diablo. So some loss in protection would be incurred anyway. In our future work, we plan to study to what extent Diablo's current overly conservative clustering can be refined by building on more advanced control flow analysis, e.g., through the use of a TSL [31].

For all benchmarks, we also observed several smaller clusters of 2–8 manually-written system library procedures that get clustered because of interprocedural gotos between them, e.g., where tail-call optimization was used. This clustering also reduces the effectiveness of the protection, but it can obviously be avoided by skipping this type of code optimization.

4.1.3 Memory Store Verification

Our prototype tool was able to protect all of the 17222 individual store instructions in our ten benchmark programs. So again, the sample size is big enough to draw conclusions on the protection's overhead in Section 4.2.3.

4.1.4 Loop Iteration Counter Duplication

Our benchmarks contain a total of 224 natural loops of which the iteration count can be determined before the loop is executed, and that contain no function calls. 100% of those are protected. 115 are inner loops in which no register pressure was detected (see Section 3.2.4); 56 are inner loops with register pressure; 11 are non-inner loops without register pressure; 15 loops are non-inner loops with register pressure. Here, "non-inner" denotes a loop has at least one nested loop.

These numbers are obviously lower than the number of branches or stores protected before, but they are still sufficient to draw more general conclusions.

4.2 Overhead of the automation at link time

This section presents the measured overhead of applying the implemented hardening techniques at link time. The overhead we are interested in is not the overhead of the hardening techniques themselves, but the additional overhead of applying them in a link-time tool that, compared to manual code hardening specialists

or hardening compilers, may lack the necessary capabilities and analysis precision to apply the hardening techniques optimally with regard to code size or performance overhead. In other words, we are interested in the potentially avoidable overhead that our prototype has introduced.

As discussed in Section 3.2.1, computing the exact additional overhead introduced by our tool compared to an optimally applied protection is impossible without implementing such an optimal protection, for which we lack the time and resources. So instead we have measured an upper bound on the overhead by carefully considering, for each hardening technique, which inserted instructions form the unavoidable, inherent implementation of the hardening protection, and which inserted instructions might have been avoided in case of a manual or compiler-based application of the hardening, or that might have been avoided with a more mature tool than our prototype. How we distinguish the categories of inserted instructions was discussed in detail in Section 3 for each of the hardening techniques. As the processors used in smart cards are simple in-order processors, counting dynamic instruction counts (with QEMU) is an acceptable approximation of the real execution time overhead.

Figures 11, 12, 14, and 15 present the results of these measurements by means of profiling. For each hardening technique, the charts present the total overhead of its application at all program points where it could be applied. Furthermore, the charts show how much of the overhead is potentially avoidable.

Overall, the visualized overheads are pretty large, but we should remind the reader that we blindly applied the protections to the whole benchmarks. In reality, smart card developers will likely limit their application to the sensitive parts of their applications.

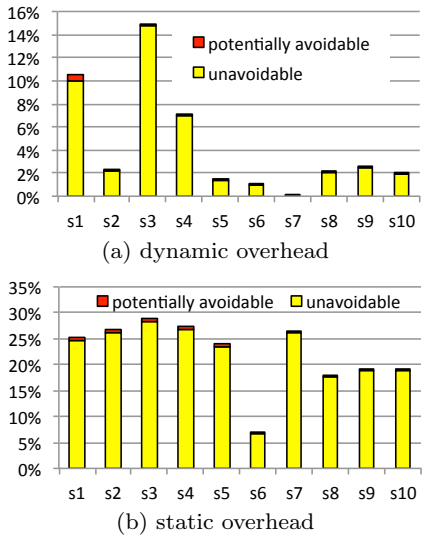


Fig. 11 Relative instruction count increases for conditional branch duplication.

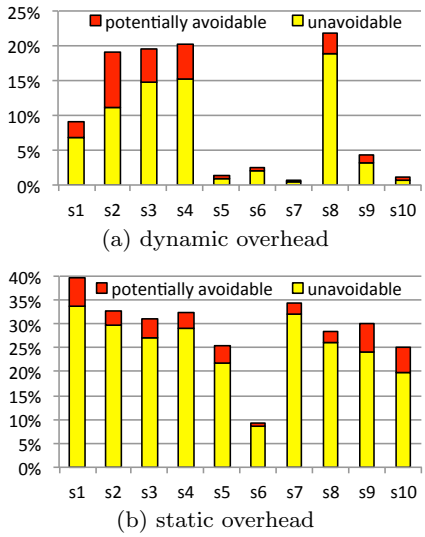


Fig. 12 Relative instruction count increases for call graph integrity checking.

4.2.1 Conditional Branch Duplication

From Figure 11, it is clear that most of the branches are protected without any overhead beyond the protection itself. Potentially avoidable overhead is at worst 0.48% for performance, and 0.60% for code size inflation.

4.2.2 Call Graph Integrity

For call graph integrity checking, we observe in Figure 12 that the potentially avoidable overhead is much higher in terms of performance overhead and in terms of code size. The reasons for this overhead differ from benchmark to benchmark.

For basicmath_small (s1), the reason is the benchmark's floating-point (FP) nature. All FP operations in the source code are compiled to invocations of hand-written FP emulation procedures. Each of them hence has a large number of calling contexts, as a result of which no free registers are found on entry/exit of these functions. So a spilling overhead needs to be paid at all their call-sites, which is unrealistically counted as avoidable. This explains why a large fraction of the static overhead is reported as avoidable. As these functions are executed very frequently, this also explains the large dynamic overhead that is, unrealistically, counted as avoidable. Obviously, in real smart card code, such FP-emulation can be expected to be quite rare. Furthermore, this result in fact demonstrates that when protecting complex, optimized hand-written assembly

code, large overheads are generally unavoidable in practice regardless of the technique used.

For bitcnts (s2), qsort_large (s3), qsort_small (s4), bf (s8), cjpeg (s9), and djpeg (s10), a large part of the total dynamic overhead is also counted as avoidable. In all cases, the overhead is due to the use of procedure pointers. We group all procedures that are potentially invoked through a procedure pointer into one cluster of which it is conservatively assumed that all argument registers (`r0`–`r3`), callee-saved registers (`r4`–`r7`), and all return value registers (`r0`–`r3`) are live on entry and/or exit. Hence, there are no free, generally accessible registers to store the cluster identifiers for these procedures. Consequently, the entry and exit identifiers passed into and out of those procedures are passed via `r8`, which cannot be accessed directly in most Thumb instructions. The resulting overhead is incurred every time an indirect call is executed, but also whenever a direct call to a procedure in this cluster is performed. We refer to the latter as a "mixed" call.

Figure 13 shows that several benchmarks feature a very high frequency of indirect and mixed calls, which explains almost all of the potentially available dynamic overhead. In bitcnts (s2), the indirect calls originate from the benchmark's main loop that invokes five different bitcounting implementations through a pointer. These comprise almost 50% of all calls, with the other 50% consisting almost entirely of recursive calls within those implementations. Those recursive calls are direct, but mixed calls, so also they come with overhead. For qsort_large (s3) and qsort_small (s4), two benchmarks centered around the `qsort` procedure, the comparator procedure is passed to `qsort` as a procedure pointer, and then invoked for pairwise comparisons between elements to be sorted. For bf (s8), the large number of mixed calls in Figure 13 comes from invoking the `fputc` procedure more than 300k times. This procedure appears in a procedure pointer list in the standard library, so even though it is invoked directly in this benchmark, a price has to be paid because it is clustered with other procedures. Finally, for cjpeg (c9) and djpeg (c10), relatively few procedure calls are executed, hence the low total dynamic overhead of this protection. But of those calls executed, a significant number are indirect, as explained in Section 4.1.2.

In real smart card applications, that mainly perform cryptographic procedures like in the benchmarks s6–s7, the prevalence of procedure pointers, indirect procedure calls, and inner-loop invocations of `putc` will typically be much lower. As can be seen for these benchmarks in Figure 12, the potentially avoidable dynamic overhead is then also much smaller.

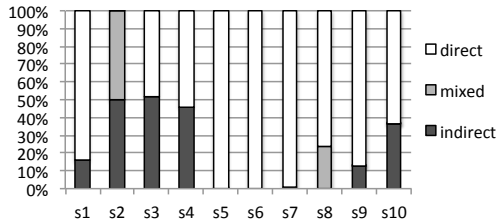


Fig. 13 Fraction of all executed procedure calls that are indirect (i.e., through procedure pointers), direct, and mixed (i.e., a direct call to a procedure that can potentially also be invoked indirectly).

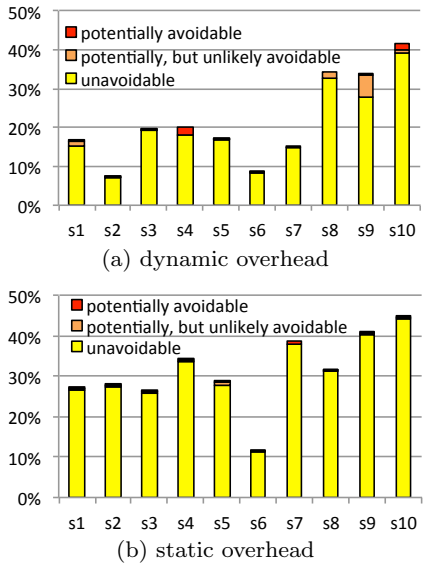


Fig. 14 Relative instruction count increases for memory store duplication.

4.2.3 Memory Store Verification

For the hardening of stores to memory, Figure 14 also depicts the fraction of the overhead that is potentially, but unlikely avoidable, as explained in Section 3.2.3. We clearly observe that almost all overhead is inherent to the protection: For 8 out of 10 benchmarks, the potentially avoidable overhead is less than 0.15%.

Only for the qsort_small (s4) and djpeg (s10) benchmarks, our accounting method of Section 3.2.3 considers a relatively large fraction of the overhead as potentially avoidable. But even in these cases, the potentially avoidable dynamic overhead is limited to at most 2.07%. We manually investigated these cases and observed that in qsort_small, the overhead is introduced in the `qsort` procedure itself, where it is indeed avoidable. However, because of the indirect call to the compare method and our tool's (overly) conservative handling of indirect calls as potentially reentrant, our tool could not avoid the overhead.

In djpeg, the overhead relates to the protection of two stores in the single inner loop of the procedure `h2v2_fancy_upsample()`. This loop has 6 variables that are live over the whole loop and are allocated to 6 registers. Furthermore, the value to be stored by each of the two store instructions also needs to occupy one register. So in this loop, at most one register from `r0` to `r7` can be available to reload the stored value into and to compare the reloaded value to the stored value. In this loop, however, the compiler has applied common subexpression elimination (CSE) to avoid having to recompute a value in between the two stores. The value of this common subexpression is stored in that one available register. As a result, our prototype does not find a free register anymore. Instead it frees one by inserting a push and a pop instruction, which are counted as potentially avoidable. The only way in which a compiler implementing the hardening could have avoided the need for such spilling, was by not applying the CSE. But then the compiler would have to insert instructions to recompute the value, which in this case also would require two instructions. So in summary, the one free register that is not strictly needed to implement the original loop without spilling, can be used either for optimizing the loop with CSE, or for implementing the hardening without additional spilling overhead. Since both options exclude each other, two extra instructions are needed in the loop anyway. As such, the overhead our accounting method considered as potentially avoidable for this benchmark, is in fact not avoidable at all.

We can therefore conclude that applying the memory store verification by means of a link-time rewriter typically introduces very little avoidable overhead.

4.2.4 Loop Iteration Counter Duplication

Figure 15 first of all shows that our limited application of loop counter duplication (excluding loops with procedure calls) did not cover the hottest loops in all benchmarks. In several benchmarks, however, the hottest, inner loops were actually protected.

For only one, qsort_small (s4), we observe a significant, potentially avoidable performance overhead. But even in that benchmark, the potentially avoidable overhead is limited to 2.07%. In fact, this is the exact same avoidable overhead as observed for qsort_small for memory store verification, because it concerns the same inner loop in the `qsort` procedure around which a register needs to be freed for both types of protections.[12]

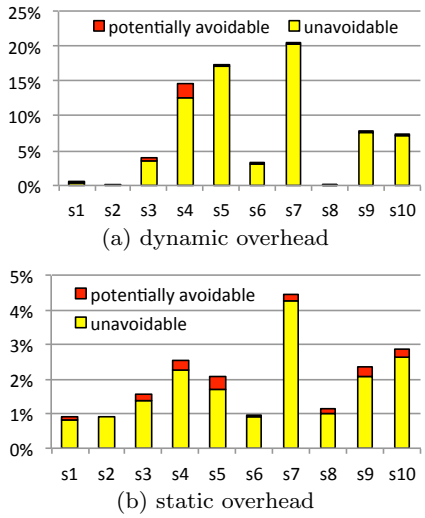


Fig. 15 Relative instruction count increase for the loop iteration counter duplication.

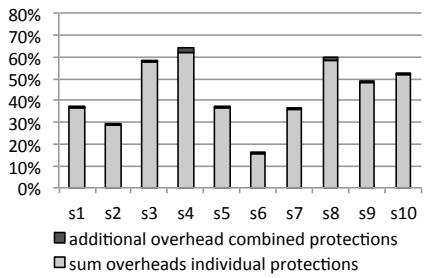
Because this protection is only applied to a limited number of loops, unlike the protection of very frequently occurring conditional branches or store operations, the static avoidable overhead is also very small, at most 0.38%. So again, we can conclude that overall, applying the protection at link time does not introduce significant avoidable overhead.

4.2.5 Combined transformations

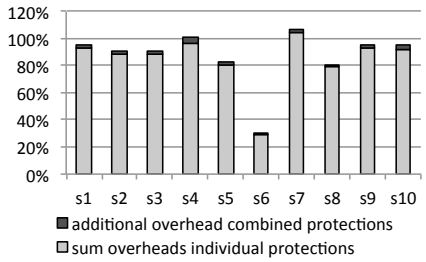
To some extent, all of the implemented protections fight for the same free registers. This is particularly the case in loops, where the live ranges of the duplicated iterators can overlap with the live ranges of temporary registers needed to implement the other protections. On the other hand, a freed register can potentially be used for multiple of those other protections.

The light bars in Figure 16 show the summed overhead of all the protections applied in isolation, i.e., the accumulated overheads of figures 11, 12, 14, and 15. The dark bars on top show the additional overhead when applying all protections together, in the following order: iteration counter duplication, conditional branch duplication, memory store verification, and call graph integrity checking. On average, the dynamic overhead of the combined protections is 0.76% higher than the sums of the individual overheads. The maximal difference is 2.24%. For the static overhead, the average difference is 2.29%, and the maximal difference is 3.74%. These low numbers demonstrate that developers can experiment with our tool's individual protections first, e.g., to find the appropriate trade-off between level of

[12] In qsort_large, this loop (which swaps a pair of elements in the array to be sorted) is also present, as it is linked from the standard C library. But in qsort_large, that loop is not hot because the elements to be swapped are very small, whereas in qsort_small the elements are quite long strings.



(a) dynamic overhead



(b) static overhead

Fig. 16 Additional overhead from combining all protections.

protection and overhead. They can then combine them afterwards, without having to fear that combining them will lead to significantly different trade-offs.

4.3 Assistance for semi-automated protection

In the preceding evaluation we have demonstrated that the vast majority of our automated link-time applications of protection policies introduces no or very little avoidable overhead. But we have also observed that in a limited number of cases, there is a significant amount of avoidable overhead. Even in those cases, a tool like ours can still be very useful to aid the developer with manually protecting the code. For example, for the protected inner loop of the `h2v2_fancy_upsample()` procedure discussed in Section 4.2.3, our tool and the open-source tool dot (http://www.graphviz.org/) can produce the annotated CFG depicted in Figure 17. It shows the inserted code, indicating which parts are potentially avoidable and which parts are not, as well as the computed liveness information and the execution counts determined through profiling. Such pictures are the perfect starting point for a developer to study whether the potentially avoidable overhead can indeed be avoided, and if so, how.

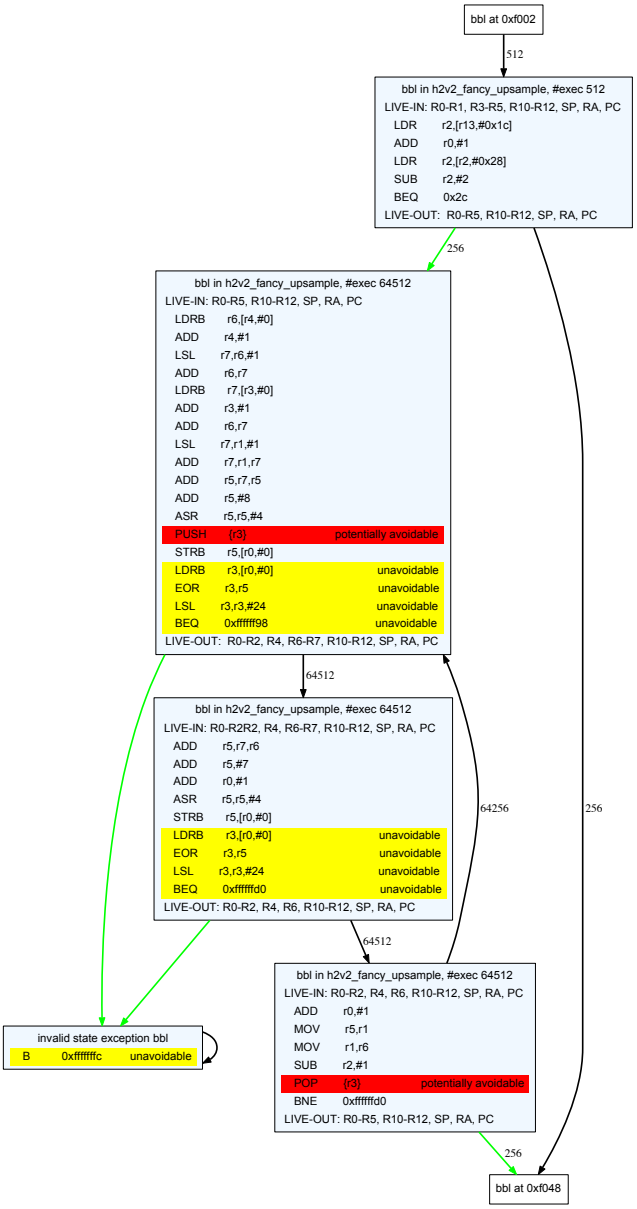


Fig. 17 Fragment of a CFG produced by our tool, annotated with execution counts and liveness information. The original instructions of the program are shown on a light gray background. The instructions highlighted in yellow and marked as "unavoidable" constitute the inherent implementation of memory store protection. The instructions highlighted in red and marked as "potentially unavoidable" constitute the additional overhead of the protection, which might have been avoidable if, e.g., the protections would be implemented by a compiler instead of a link-time rewriter.

5 Related Work

As cited in the introduction, many protection schemes have been proposed to mitigate fault injection attacks. In this paper, we focused on an automated approach to deploy existing protections on native code. By deploying the techniques in a link-time rewriter, we avoid the need to interfere with the operation of existing compilers or with an application's source code development. While the deployment of simple protections at link time, by duplicating code like we do here, was already presented in an earlier publication [35], this paper is the

first to evaluate the additional (i.e., potentially avoidable) overhead caused by their link-time deployment, and to include more global protections such as the loop iteration counter protection.

As discussed in Section 2, other existing static rewriting techniques cause too much overhead when applied to smart card code. However, run-time rather than static approaches have also been proposed to achieve similar advantages in the context of bytecode execution on Java smart cards. Lackner et al. proposed to adapt the virtual machine (VM) that interprets the bytecode to inject the necessary redundancy in the executed code [27, 28]. By duplicating code at run time, they achieve the same goal of providing protection without interfering with the software development cycle. Additionally, they proposed hardware extensions that the VM can exploit to reduce the overhead of the protections to near zero.

6 Conclusions and Future Work

On the basis of demonstrated capabilities of the Diablo link-time rewriting framework, we argued for its capabilities in development scenarios centered around black-box, third-party compiler tool flows.

Using Diablo, we implemented a prototype tool that automates protection against single fault injection attacks. On code generated with industrial-strength, proprietary compilers, we measured an upper bound of the additional overhead that such an automated tool introduces on top of the inherent overhead of the protections. We observed that the overhead potentially caused by the nature of our approach and tool, both in terms of code size and of execution time, is very limited. Therefore we are convinced that a link-time rewriter can apply the protections (almost) as well as a manual assembler rewriter can, or as a compiler can. We can therefore conclude that automated, link-time, single-fault-injection protection is a realistic, promising direction, which can provide a significant step forward towards more productive smart card programming and code hardening.

Future R&D includes better mechanisms to select where to apply the transformations, including possibilities for steering the protection process through an interactive, GUI-based binary code hardening tool. In addition, we plan to research how to specify transformations without having to hard-code them in a tool, but by instead using convenient APIs like those of existing code instrumentation tools [16]. Finally, it would be useful to study whether our approach would incur more overhead when applying protection schemes against multiple fault injection attacks. In case those schemes need more free registers, the overhead of applying them at link-time might well be higher than what was demonstrated with our current single fault protection. Furthermore, we expect that it will be more difficult to distinguish between inherent overhead of those schemes, and additional overhead following from our approach.

References

1. M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.
2. P.E. Ammann and J.C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Trans. Comp.*, 37(4):418–425, 1988.
3. K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proc. 8th ACM European Conf. on Computer Systems*, pages 295–308, 2013.
4. B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. In *ACM QoP*, pages 15–20, 2007.
5. B. Anckaert, F. Vandeputte, B. De Bus, B. De Sutter, and K. De Bosschere. Link-time optimization of IA64 binaries. In *Proc. Euro-Par*, pages 284–291, 9 2004.
6. C. Aumüller, P. Bier, W. Fischer, P., and J.-P. Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In *Proc. CHES*, pages 260–275, 2002.
7. A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11(12):1491–1501, 1985.
8. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. Cryptology ePrint Archive, Report 2004/100, 2004.
9. F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc USENIX*, pages 41–46, 2005.
10. R. Bertran, M. Gil, J. Cabezas, V. Jimenez, L. Vilanova, E. Morancho, and N. Navarro. Building a global system view for optimization purposes. In *Proc. Workshop Interaction between Operating Systems and Computer Architecture*, 2006.
11. D. Chanet. *Memory Footprint Reduction for Operating System Kernels.* PhD thesis, Ghent University, 2007.
12. D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. Automated reduction of the memory footprint of the Linux kernel. *ACM Trans. Emb. Comp. Syst.*, 6(4):23::1–23::48, 2007.
13. H. Choukri and M. Tunstall. Round reduction using faults. In *Proc. FDTC*, pages 13–24, 2005.
14. L. Claes. Colos: een optimaliserende linker voor de superH. Master's thesis, Ghent University, 2003.
15. B. De Bus. *Reliable, retargetable and extensible link-time program rewriting.* PhD thesis, Ghent University, 2005.
16. B. De Bus, D. Chanet, B. De Sutter, L. Van Put, and K. De Bosschere. The design and implementation of FIT: a flexible instrumentation toolkit. In *Proc. ACM PASTE*, pages 29–34, 2004.
17. B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Trans. Prog. Lang. and Syst.*, 27(5):882–945, 2005.

18. B. De Sutter, B. De Bus, and K. De Bosschere. Bidirectional liveness analysis, or how less than half of the Alpha's registers are used. *Journal of Systems Architecture*, 52(10):535–548, 2006.
19. B. De Sutter, L. Van Put, D. Chanet, B. De Bus, and K. De Bosschere. Link-time compaction and optimization of ARM executables. *ACM Trans. Emb. Comp. Syst.*, 6(1):5::1–5::43, 2007.
20. S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Proc. ACM POPL*, pages 12–24, 1998.
21. E. N. Dolgova and A. V. Chernov. Automatic reconstruction of data types in the decompilation problem. *Program. Comput. Softw.*, 35(2):105–119, March 2009.
22. M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. IEEE WWC-4*, pages 3–14, 2001.
23. W.C. Huffman and V. Pless. *Fundamentals of error-correcting codes.* Cambridge university press, 2003.
24. M. Karpovsky, K.J. Kulikowski, and A. Taubin. Robust protection against fault-injection attacks on smart cards implementing the advanced encryption standard. In *Proc. Int'l Conf. on Dependable Systems and Networks*, pages 93–101, June 2004.
25. C.H. Kim and J.-J. Quisquater. Fault attacks for CRT based RSA: new attacks, new results and new countermeasures. In *Proc. WISTP*, pages 215–228, 2007.
26. J. Kinder, F. Zuleger, and H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proc. 10th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 214–228, 2009.
27. M. Lackner, R. Berlach, M. Hraschan, R. Weiss, and C. Steger. A defensive java card virtual machine to thwart fault attacks by microarchitectural support. In *Proc. Int'l Conf on Risks and Security of Internet and Systems (CRiSIS)*, pages 1–8, Oct 2013.
28. M. Lackner, R. Berlach, W. Raschke, R. Weiss, and C. Steger. A defensive virtual machine layer to counteract fault attacks on java cards. In *Information Security Theory and Practice. Security of Mobile and Cyber-Physical Systems*, volume 7886 of *Lecture Notes in Computer Science*, pages 82–97. 2013.
29. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. 2004 Int'l Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
30. J.R. Levine. *Linkers and Loaders.* Morgan Kaufmann Publishers Inc., 1999.
31. J. Lim and T. Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *ACM Trans. Program. Lang. Syst.*, 35(1):4:1–4:59, April 2013.
32. M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *ACM DRM*, pages 75–82, 2005.
33. M. Madou, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. Link-time optimization of MIPS programs. In *Proc. ESA*, pages 70–75, 2004.
34. M. Madou, L. Van Put, and K. De Bosschere. Loco: an interactive code (de)obfuscation tool. In *Proc. PEPM '06*, pages 140–144, 2006.
35. J. Maebe, R. De Keulenaer, B. De Sutter, and K. De Bosschere. Mitigating smart card fault injection with link-time code rewriting: a feasibility study. In *Proc. 17th Int'l Conf. on Financial Cryptography and Data Security*, 2013.
36. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security).* Springer-Verlag, 2007.
37. C. Markantonakis, K. Mayes, M. Tunstall, D. Sauveron, and F. Piper. Smart card security. In *Computational Intelligence in Information Assurance and Security*, pages 201–233. Springer, 2007.
38. S.S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.
39. N. Oh, S. Mitra, and E.J. McCluskey. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Trans. Comput.*, 51(2):180–199, February 2002.
40. P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A.D. Keromytis. Retrofitting security in COTS software with binary rewriting. In *Proc. 26th IFIP TC 11 Int'l Information Security Conf.*, pages 154–172, 2011.
41. B. Randell. System structure for software fault tolerance. In *Proc. of the international conference on Reliable software*, pages 437–449, 1975.
42. M. Rebaudengo, M.S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. In *Proc. IEEE SCAM*, pages 33–42, 2001.
43. G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. SWIFT: Software implemented fault tolerance. In *Proc. ACM CGO*, pages 243–254, 2005.
44. A.A. Sere, J. Iguchi-Cartigny, and J.-L. Lanet. Automatic detection of fault attack and countermeasures. In *Proc. of the 4th Workshop on Embedded Systems Security*, pages 7:1–7:7, 2009.
45. M. Smithson, K. Anand, A. Kotha, K. Elwazeer, N. Giles, and R. Barua. Binary rewriting without relocation information. Technical report, University of Maryland, nov 2010.
46. W. Torres-Pomales. Software fault tolerance-tutorial, 2000. NASA/TM-2000-210616.
47. E. Trichina and R. Korkikyan. Multi fault laser attacks on protected crt-rsa. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pages 75–86, Aug 2010.
48. L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *Proc. ISSPIT*, pages 7–12, 2005.
49. L. Van Put, B. De Sutter, M. Madou, B. De Bus, D. Chanet, K. Smits, and K. De Bosschere. LANCET: a nifty code editing tool. In *ACM PASTE*, pages 75–81, 2005.
50. R. Wartell, V. Mohan, K.W. Hamlen, and Z. Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proc. 28th Annual Computer Security Applications Conf.*, pages 299–308, 2012.
51. D. Williams, W. Hu, J.W. Davidson, J.D. Hiser, J.C. Knight, and A. Nguyen-Tuong. Security through diversity: leveraging virtual machine technology. *IEEE Security & Privacy*, 7(1):26–33, 2009.
52. J. Yiu. *The Definitive Guide to the ARM Cortex-M0.* Newnes, 2011.
53. M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proc. Usenix Security*, 2013.
54. L. Zhao, G. Li, B. De Sutter, and J. Regehr. ARMor: Fully verified software fault isolation. In *Proc. EMSOFT*, pages 289–298, 2011.